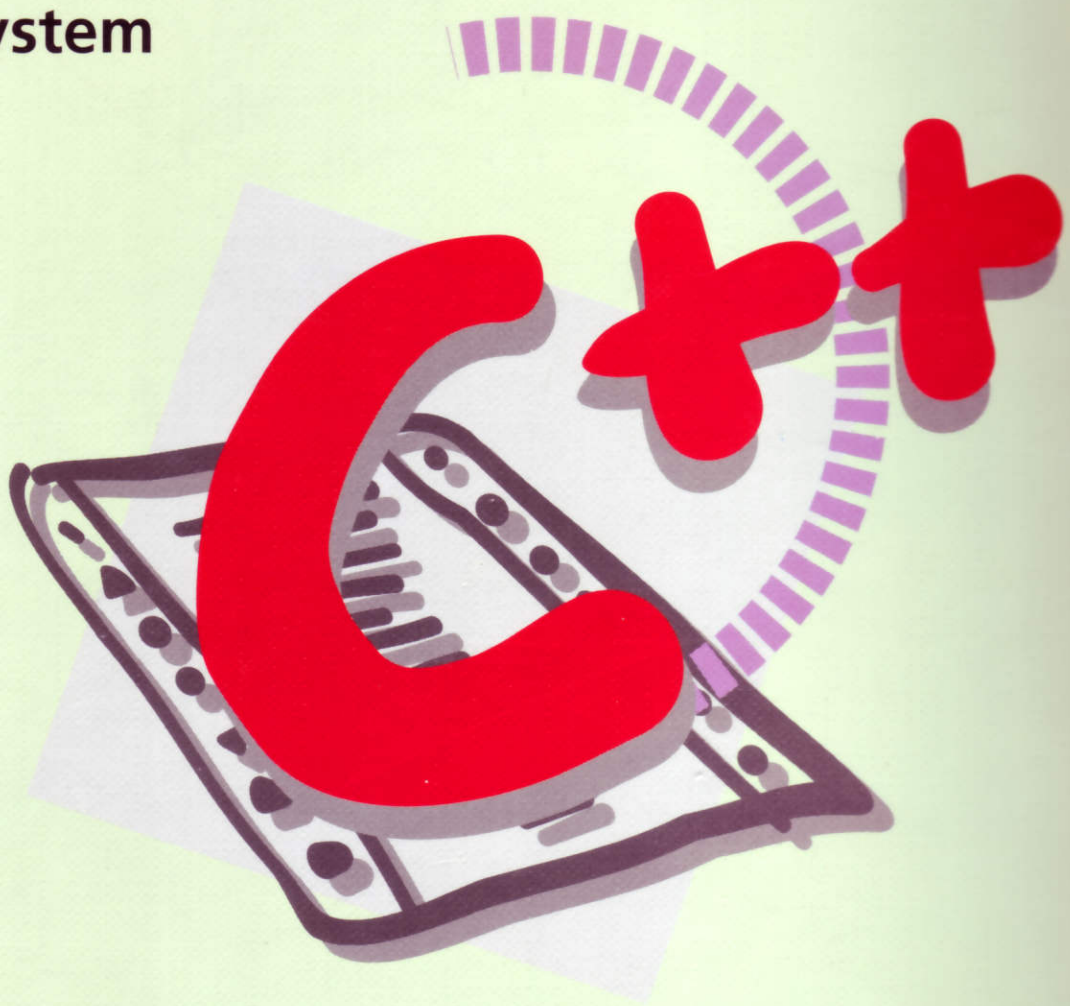
Maxon C++ 3

Integriertes Compilersystem



AMIGA®

MAXON computer

MaxonC++ Version 3.1

Wichtige Information: Bitte sofort lesen!

Sehr geehrter Kunde,

Sie haben von uns die neueste Version 3.1 von MaxonC++ erhalten. In dieser wurde der bisherige Editor *EDWARD* durch den neuen Editor *MaxonED* ersetzt, der wesentlich mehr Funktionen bietet, wie z.B. Farbhervorhebungen im Quelltext.

Durch diese Änderung sind die Teile des Handbuchs, in denen *EDWARD* behandelt wurde, überflüssig geworden. In Ihrem Handbuch fehlen dadurch die Seiten 85-156 und 187-200. Sie werden in den anderen Teilen des Handbuchs Verweise auf *EDWARD* finden, die allerdings größtenteils auch für *MaxonED* gelten. An der beschriebenen Zusammenarbeit zwischen Editor und Compiler hat sich prinzipiell nichts geändert.

Das Handbuch für *MaxonED* finden Sie nach der Installation von MaxonC++ als AmigaGuide-Datei im Verzeichnis „MCPP:BIN".

Es handelt sich dabei um eine Übergangslösung, die nur bis zum Erscheinen der Version 4 Ende 1996 beibehalten wird.

Senden Sie bitte unbedingt Ihre Registrierungskarte ein, damit wir Sie rechtzeitig über die neue Version und die Updatemöglichkeiten informieren können.

Ihre MAXON Computer GmbH

AMIGA

MaxonC++

Integriertes C++-Compilersystem

Developer Version

MAXON computer

MaxonC++ 3 Developer

Autoren:

Compiler:	**Jens Gelhar**
Editor:	**Thomas E. Wieger**
HotHelp 3:	**Hartmut Stein**
Easy-Objects:	**Jochen Becher**
Assembler:	**Klaus Dieter Sommer**
Debugger:	**Jochen Becher**

© 1992-95 MAXON Computer GmbH

Alle Rechte vorbehalten. Dieses Handbuch und die dazugehörige Software ist urheberrechtlich geschützt. Es darf in keiner Form (auch auszugsweise) mittels irgendwelcher Verfahren reproduziert, gesendet, vervielfältigt bzw. verbreitet oder in eine andere Sprache übersetzt werden.

Bei der Erstellung des Programmes, der Anleitung sowie Abbildungen wurde mit allergrößter Sorgfalt vorgegangen. Trotzdem können Fehler nicht ausgeschlossen werden. MAXON übernimmt keinerlei Haftung für Schäden, die auf eine Fehlfunktion von Programmen zurückzuführen sind.

Alle Informationen, die in der vorliegenden Anleitung enthalten sind, werden ohne Rücksicht auf einen eventuellen Patentschutz veröffentlicht. Ebenso werden Warenzeichen ohne Gewährleistung einer freien Verwendung benutzt.

Support

Bei Fragen zu **MaxonC++ 3** wenden Sie sich bitte an:

MAXON Computer GmbH
"**MaxonC++ 3** Support"
Postfach 5969
65734 Eschborn

Copyrights und Warenzeichen:
Commodore® und Amiga® sind eingetragene Warenzeichen der Commodore-Amiga Inc.
AmigaDOS™, Kickstart™, Intuition™ und Workbench™ sind Warenzeichen der Commodore-Amiga Inc.
Amiga Workbench™ ist ein Copyright der Commodore Amiga Inc.
MAXON® ist ein eingetragenes Warenzeichen der MAXON Computer GmbH

AMIGA

MaxonC++

Integriertes C++-Compilersystem

MAXON
computer

MaxonC++ 3

Autoren:

Compiler:	**Jens Gelhar**
Editor:	**Thomas E. Wieger**
HotHelp 3:	**Hartmut Stein**
Easy-Objects:	**Jochen Becher**
Assembler:	**Klaus Dieter Sommer**
Debugger:	**Jochen Becher**

© 1992-94 MAXON Computer GmbH

Alle Rechte vorbehalten. Dieses Handbuch und die dazugehörige Software ist urheberrechtlich geschützt. Es darf in keiner Form (auch auszugsweise) mittels irgendwelcher Verfahren reproduziert, gesendet, vervielfältigt bzw. verbreitet oder in eine andere Sprache übersetzt werden.

Bei der Erstellung des Programmes, der Anleitung sowie Abbildungen wurde mit allergrößter Sorgfalt vorgegangen. Trotzdem können Fehler nicht ausgeschlossen werden. MAXON übernimmt keinerlei Haftung für Schäden, die auf eine Fehlfunktion von Programmen zurückzuführen sind.

Alle Informationen, die in der vorliegenden Anleitung enthalten sind, werden ohne Rücksicht auf einen eventuellen Patentschutz veröffentlicht. Ebenso werden Warenzeichen ohne Gewährleistung einer freien Verwendung benutzt.

Support

Bei Fragen zu **MaxonC++ 3** wenden Sie sich bitte an:

MAXON Computer GmbH
"**MaxonC++ 3** Support"
Postfach 5969
65734 Eschborn

Copyrights und Warenzeichen:
Commodore® und Amiga® sind eingetragene Warenzeichen der Commodore-Amiga Inc.
AmigaDOS™, Kickstart™, Intuition™ und Workbench™ sind Warenzeichen der Commodore-Amiga Inc.
Amiga Workbench™ ist ein Copyright der Commodore Amiga Inc.
MAXON® ist ein eingetragenes Warenzeichen der MAXON Computer GmbH

MaxonC++ 3 ist da!

Nachdem die Welt der professionellen Softwareentwicklung die stille Revolution bereits erlebt hat, hat sich auch das Konzept der objektorientierten Programmierung sich in allen Bereichen durchgesetzt.

Nachdem Simula in den späten 60er Jahren ihrer Zeit zu weit voraus war und Smalltalk später zu sehr mit allen vertrauten Konzepten der Programmierung brach, um sich durchsetzen zu können, gibt es mit MaxonC++ eine objektorientierte Programmiersprache, die sich als der neue Standard in der Softwareentwicklung etabliert hat.

Es gibt viele Gründe, die gerade für C++ sprechen. Zunächst setzt diese Sprache auf C auf, der im professionellen Bereich bisher verbreitetsten Sprache. Das geht so weit, daß C-Quelltexte sich mit minimalen oder ganz ohne Modifikationen mit einem C++-Compiler übersetzen lassen. Diese Abwärtskompatiblität erleichtert natürlich den Umstieg von C nach C++ gewaltig und macht C++ für jeden C-Programmierer zur ersten Wahl. Ferner wurde C++ konsequent so konzipiert, daß eine objektorientierte Lösung weder beim Speicherplatz noch in Bezug auf die Laufzeit einer konventionellen Lösung unterlegen ist und so die Effizienz, die man von C gewohnt ist, erhalten bleibt.

Nicht zuletzt ist zu erwähnen, daß C++ von den AT&T Bell Laboratories entwickelt wurde und von AT&T, also gewiß keiner kleinen Firma, gefördert wird. Schon allein dieser Umstand hat für die Verbreitung von C++ im UNIX-Bereich und bei den sich mehr und mehr am Workstation-Markt orientierenden Microcomputer gesorgt.

Nur in der Amiga-Welt war von all diesen aufregenden Entwicklungen lange Zeit nichts zu spüren. Während auf anderen Systemen sogar jeder bessere C-Compiler objektorientierte Konzepte von C++ übernahm oder gleich zu einem vollständigen C++-System ausgebaut wurde, empfand der Amiga-User schon das Erscheinen von ANSI-kompatiblen C-Compilern als Fortschritt. Da leistungsfähige Compiler aber die Voraussetzung für professionelle Programmentwicklung sind, litt unter diesem Mangel nicht nur der persönliche Komfort der Amiga-Programmierer, sondern auch die Entwicklung der Amiga-Software allgemein wurde behindert und drohte den Anschluß an konkurrierende Systeme endgültig zu verlieren. Es war höchste Zeit, daß etwas geschah.

Nach mehr als dreijähriger Entwicklungszeit und einer umfangreichen Debugging-Phase ist MaxonC++ 3 jetzt einsatzbereit. MaxonC++ ist keine Portierung von einem anderen System, sondern wurde speziell für den Amiga (und auf einem Amiga) entwickelt und auf die Besonderheiten dieses Systems zugeschnitten. Der Compiler ist zum „3.0"-Standard (Light-Version zum 2.0-Standard) kompatibel und erzeugt aus dem C++-Quelltext direkt ausführbaren Code, ohne Umwege über C, Assembler oder andere Sprachen. Trotzdem läßt er sich auf ANSI-C-Emulation umschalten, um volle Kompatiblität zu den vorhandenen C-Compilern zu erreichen, und kann optional auch Assembler-Quelltext für die Weiterverarbeitung mit einem fremden Assemblern erzeugen.

Lieferumfang

Das MaxonC++ Compilersystem besteht aus zwei Handbüchern, die durch Einlageblätter voneinander getrennt sind:

1. MaxonC++ Anwender-Handbuch
   Compilersystem, Make, Linker, CLI-Verion, Editor und Anhänge
2. MaxonC++ Tutorial – Einführung in C und C++

Die Developer-Version des Compilersystems ist umweitere Handbücher ergänzt:

3. Tutorial-Ergänzung – Exceptions, Templates
4. MaxonC++ Referenz-Handbuch
5. Maxon HotHelp 3 – Online-Hilfe
6. MaxonEASY-OBJECTS – Klassenbibliothek
7. Maxon Source-Level-Debugger
8. MaxonASM (CLI-Version)

Der Compiler und die Zusatzprogramme werden auf mehreren Disketten geliefert. Genaue Angaben zur Installation und zur Diskettenanzahl entnehmen Sie bitte der „Readme"-Datei auf der Installationsdiskette. Da sich nach der Drucklegung noch einige Änderungen, was die Diskettenanzahl und die Programmzusammenstellung angehen, ergeben können, ist dies der beste Weg, um das Handbuch so aktuell wie möglich zu halten.

Developer-Upgrade

Allen registrierten Besitzern der Light-Version bieten wir die Möglichkeit eines Upgrades auf die MaxonC++-Developer-Version. Das Upgrade beinhaltet ein neues Compilersystem, die Klassenbibliothek, den MaxonSource-Level-Debugger, die Online-Hilfe HohHelp 3 und den MaxonASM in der CLI-Verion. Die Upgradekonditionen erfragen Sie bitte bei Maxon Computer: 06196/481811.

Registrierung

Ihrem Compilersystem liegt eine Registrierkarte bei, die Sie umgehend an uns zurücksenden sollten. Sie gewährleisten dadurch, daß Sie regelmäßig über Produktneuerungen und Neuerscheinungen informiert werden. Desweiteren steht Ihnen dann unsere tägliche Hotline für Problemlösungen zur Verfügung.

Hotline und Support

Unsere Hotline ist werktags zwischen 15:00 und 17:00 Uhr unter der Rufnummer 06196/43780 zu erreichen. Bitte prüfen Sie vor einem Anruf genau den Problemfall:

- Haben Sie die Software richtig installiert?
- Benutzen Sie die richtige Workbench Version (nicht unter 2.0)?
- Laufen irgendwelche Programme im Hintergrund (Screenblanker, Maustools o.ä.)?

Bitte haben Sie Verständnis, daß wir keine Garantie für das Zusammenspiel mit anderer Software geben können. Besonders einige Tools, die in der Startup-Sequence mitgestartet werden, können die Arbeitsfähigkeit sehr stören.

Bei einem Anruf halten Sie bitte Ihre Registriernummer, die Versionsnummer des jeweiligen Softwarepaketes und eine genaue Konfigurationsbeschreibung bereit. Die Registriernummer ist auf den Orginal-Disketten aufgedruckt. Die Versionsnummer finden Sie entwender im Info-Dialogfenster des jeweiligen Programmes oder wenn Sie (ab Amiga OS 2.0) den Befehl:

Version "Programmname" full

in der Shell (CLI) eingeben.

Wenn Sie uns schreiben, benötigen wir zum zügigen Bearbeiten, ebenfalls die beschriebenen Angaben.

Richten Sie Ihr Anschreiben an folgende Adresse:

Maxon Computer GmbH
"MaxonC++ 3-Support"
Industriestraße 26
W-65760 Eschborn

Version
3.10 r1 (22.05.96)

Literaturangaben

Wenn Sie bereits in einer anderen Programmiersprache als C++ programmieren können, ist das Tutorial-Handbuch eine sehr gute Einführung für Sie, um die Syntax und den Umgang mit C++ kennenzulernen. Als Programmieranfänger empfiehlt es sich jedoch, ein Grundeinführungsbuch durchzuarbeiten. Die nachfolgende Liste enthält eine Auswahl an Büchern zu den Themen objektorientierte Programmierung im allgemeinen, Programmierung in C und C++ und zur Programmierung des Amiga Betriebssystems.

Objektorientierte Programmierung allgemein

Objektorientierte Programmierung - Eine Einführung
R. Engel - 338 Seiten - ISBN 3-89090-353-3
Markt&Technik Verlag AG/Haar

C und C++ Literatur
Lernen und Üben: C-Schulung
H. Niegemann und T. Schäfer - 354 Seiten - ISBN 3-89090-218-9
Markt&Technik Verlag AG/Haar

Die Einführung: C
S. Baloui - 379 Seiten - ISBN 3-89090-334-7
Markt&Technik Verlag AG/Haar

Programmieren in C++
E. Huckert - 348 Seiten - ISBN 3-89090-171-9
Markt&Technik Verlag AG/Haar

Clever Programmieren: C++
M. Haun - 352 Seiten - ISBN 3-88322-327-1
IWT Verlag GmbH/Vaterstetten

IWT Kompakt: C-Referenz für alle ANSI-C-Compiler
P. Prinz - 304 Seiten - ISBN 3-88322-340-9
IWT Verlag GmbH/Vaterstetten

C++ lernen und beherrschen: Einführung und Leitfaden
S. B. Lippmann - ca. 500 Seiten - ISBN 3-89319-375-8
Addison-Wesley (Deutschland) GmbH/Bonn

Die C++ Programmiersprache
B. Stroustrup - 380 Seiten - ISBN 3-89319-386-3
Addison-Wesley (Deutschland) GmbH/Bonn

The C++ Programming Language
B. Stroustrup - 576 pages - ISBN 0-201-53992-6
Addison-Wesley (Deutschland) GmbH/Bonn

The Annotated C++ Reference Manual
M. A. Ellis/ B. Stroustrup - 400 pages - ISBN 0-201-51459-1
Addison-Wesley (Deutschland) GmbH/Bonn

Die Kunst der objektorientierten Programmierung - Prinzipien und Konzepte in C++
A. Schäpers - ca. 440 Seiten - ISBN 3-89319-226-3
Addison-Wesley (Deutschland) GmbH/Bonn

C-Programmieren: Ein Kurs zum Selbststudium
G. Krüger - ca. 500 Seiten - ISBN 3-89319-372-3
Addison-Wesley (Deutschland) GmbH/Bonn

Der C-Experte: Programmieren ohne Pannen
A. Koenig - 190 Seiten - ISBN 3-89319-233-6
Addison-Wesley (Deutschland) GmbH/Bonn

Debugging C
R. Ward - 322 Seiten - ISBN 3-925118-90-X
Addison-Wesley (Deutschland) GmbH/Bonn

C Professional Edition
W. Herglotz - ca 450 Seiten - ISBN 3-89360-004-3
BHV Verlags GmbH/Korschenbroich

Das Einsteigerseminar C
W. Herglotz - 224 Seiten - ISBN 3-89360-606-8
BHV Verlags GmbH/Korschenbroich

Das Einsteigerseminar C++
W. Herglotz - 224 Seiten - ISBN 3-89360-622-X
BHV Verlags GmbH/Korschenbroich

C++ für Programmierer
F. Bause/W. Tölle - 286 Seiten - ISBN 3-528-05110-8
Friedr. Vieweg & Sohn Verlagsgesellschaft mbH/Wiesbaden

Einführung in die Programmiersprache C++
F. Bause/W. Tölle - 171 Seiten - ISBN 3-528-14689-3
Friedr. Vieweg & Sohn Verlagsgesellschaft mbH/Wiesbaden

Referenzhandbuch Standart C
P.J. Plauger/J. Brodie - 236 Seiten - ISBN 3-528-04741-0
Friedr. Vieweg & Sohn Verlagsgesellschaft mbH/Wiesbaden

C++ Einführung in die objektorientierte Programmierung
S. Prata - 750 Seiten - ISBN 3-89362-701-4
tewi Verlag

Der schnelle Umstieg auf C++
M. Aupperle - 76 Seiten (Chip Spezial) - ISBN 3-8023-1088-8
Vogel Verlag und Druck KG/Würzburg

C für Umsteiger
B. Klöppel - 122 Seiten (Chip Spezial) - ISBN 3-8023-1081-0
Vogel Verlag und Druck KG/Würzburg

Amiga Bücher

Amiga-Programmierrichtlinien
H. Gzella - 272 Seiten - ISBN 3-87791-049-1
Markt&Technik Verlag AG/Haar

Amiga ROM Kernel Reference Manual: Devices
512 pages - ISBN 0-201-56775-X
Addison-Wesley (Deutschland) GmbH/Bonn

Amiga ROM Kernel Reference Manual: Libraries
960 pages - ISBN 0-201-56774-1
Addison-Wesley (Deutschland) GmbH/Bonn

Amiga ROM Kernel Reference Manual: Includes and Autodocs
1000 pages - ISBN 0-201-56773-3
Addison-Wesley (Deutschland) GmbH/Bonn

Amiga Hardware Reference Manual
512 pages - ISBN 0-201-56776-8
Addison-Wesley (Deutschland) GmbH/Bonn

Amiga User Interface Style Guide
224 pages - ISBN 0-201-57757- 7
Addison-Wesley (Deutschland) GmbH/Bonn

MaxonHotHelp 3 OS 3.1 Dokumentationen
Deutschsprachige Online-Dokumentation Best.Nr.: 920313
Beschreibt alle Betriebssystemfunktionen unter Kickstart/Workbench 3.1
MAXON Computer GmbH Eschborn

Anwenderhandbuch

MaxonC++ 3 ist da! V-5

Lieferumfang V-6
Developer-Upgrade V-6
Registrierung V-6
Hotline und Support V-6
Literaturangaben V-8
Objektorientierte Programmierung allgemein V-8
Amiga Bücher V-11

Maxon C++ 3 3

1.1 Was gibt es Neues? 3
1.2 Der Umstieg 4
1.2.1 Konfigurationsdateien 4
1.2.2 „Edward.DEF" 4
1.2.3 Dateien und Verzeichnisse 6
1.3 Ein ersterÜberblick 7
1.3.1 Disketten 7
1.3.2 Installation 7
1.3.3 Die Komponenten des MaxonC++ Systems 7
1.4 Erste Schritte mit MCPP 9

2. Die integrierte Entwickungsumgebung 11

2.1 Programmstart und Optionen 11
2.2 Compiler und Editor - eine harmonische Beziehungskiste 12
2.3 Das Konzept 13
2.4 Compiler-Funktionen 14
2.4.1 Programme übersetzen 14
2.4.1.1 Das Compiler-Fenster 14
2.4.1.2 Abbruch und Ausstieg 16
2.4.1.3 Auswahl von Fehlermeldungen 16
2.4.1.4 Vom Compiler in den Editor 17
2.4.1.5 „Weiter" 18

2.4.1.6 Programmstart aus dem Compiler-Fenster 18
2.4.1.7 Erneutes öffnen des Compiler-Fenster 18
2.4.2 Ausführen und Abbrechen 18
2.4.3 Aufruf des Debuggers 19
2.4.4 Exe- und Objektdateien 19
2.4.5 Daten und Informationen 20
2.5 Features und Optionen 20
2.5.1 Ein kurzer Überblick 20
2.5.2 Compiler-Optionen 20
2.5.2.1 Include-Dateien 20
2.5.2.2 Vorcompilierte Header-Dateien 22
2.5.2.3 Der Arbeitsspeicher des Compilers 23
2.5.2.4 Debug-Datei 23
2.5.2.5 Vordefinieren von Symbolen 23
2.5.2.6 Diverse kleine, aber trotzdem wichtige Schalter 24
2.5.2.6.1 C++ 24
2.5.2.6.2 Unterbrechen 24
2.5.2.6.3 IntMult 32 24
2.5.2.6.4 Debug-Datei 25
2.5.2.6.5 Symbol-Hunk 25
2.5.2.6.6 Asm schreiben 25
2.5.2.6.7 Source in Asm 26
2.5.2.6.8 Optimieren 26
2.5.2.6.9 Code für höhere Prozessorgenerationen 26
2.5.2.6.10 Strategien für Template-Funktionen 27
2.5.2.6.11 Exceptions 28
2.5.2.7 Optionale Warnungen und Meldungen 28
2.5.3 Der Linker-Requester 30
2.5.3.1 Datenmodelle 30
2.5.3.2 Linken oder nicht Linken - das ist hier die Frage 31
2.5.3.3 Verhalten bei Fehler 32
2.5.3.4 Bibliotheken 32
2.5.3.5 Zusammenfassen von Hunks 32
2.5.3.6 Schreiben von Dateien 33
2.5.4 MCPP-Systemeinstellungen 34
2.5.4.1 Erzeugen von Icons 34
2.5.4.2 Abhängigkeiten in Make-Dateien 34
2.5.4.3 Protokollieren von Ein- und Ausgaben 34
2.5.4.4 Argumente aus der imaginären Kommandozeile 34
2.5.4.5 Einbindung des Maxon Assemblers 35

2.5.4.6 Debugger 35
2.5.4.7 Stackgröße 35
2.5.4.8 Autosave 35
2.5.4.9 Dateiauswahlbox 36
2.5.5 Abspeichern der Konfigurationsdatei 36
2.5.5.1 Allgemeines 36
2.5.5.2 Der Editor 36
2.5.5.3 Fensterdaten 37
2.5.5.4 Switches und Schalter 37
2.5.5.5 Stringoptionen 37
2.6 Die ARexx-Schnittstelle 38
2.6.1 Grundlagen 38
2.6.2 Der ARexx-Port 38
2.6.3 Die ARexx-Befehle 39
2.6.3.1 Schließen, öffnen und Beenden 39
2.6.3.2 Übersetzen, Starten und Ähnliches 39
2.6.3.3 Requester und Informationen 40
2.6.3.4 Make 40
2.6.3.5 Schreiben von Dateien 40
2.6.3.6 Optionen speichern und setzen 40
2.6.3.7 Informationen über die Entwicklungsumgebung 41
2.6.3.8 Fehlermeldungen und Fehlerstellenansprung 41
3. MAKE - eine integrierte Projektverwaltung 43
3.1 Das Prinzip 43
3.1.1 Wozu eine Projektverwaltung? 43
3.1.2 Was ist das überhaupt - ein Projekt? 44
3.1.3 Was hat das ganze mit „Make" zu tun? 45
3.2 Die Praxis 45
3.2.1 Erstellen von Projekten 45
3.2.2 Hinzufügen von Dateien, umständliche Methode 46
3.2.3 Übersetzen des Projekts 48
3.2.4 Hinzufügen von Dateien, einfache Methode 48
3.2.5 Abhängigkeiten 48
3.2.6 Weitere Funktionen im Make-Fenster 49
3.2.6.1 Projektpflege 49
3.2.6.2 Übersetzung erzwingen 50
3.2.6.3 Dateien in den Editor laden 50
3.2.6.4 Projektdatei wechseln 50

3.2.7 Compileranweisungen für Objektdateien 50
3.2.7.1 Interne Übersetzung 50
3.3 Handgemachte Makedateien 52
3.3.1 Die Grundlagen 52
3.3.2 Der Linkeraufruf 52
3.3.3 Internes Compilieren 53
3.3.4 Externe Übersetzung 53
3.3.5 Weitere Features 54
3.3.5.1 Zeilenverbindung 54
3.3.5.2 Kommentare 54
3.3.5.3 Makros 55
3.3.5.4 Regeln 56
4.1 Prinzipielles 59
4.2 Standard-Funktionen und das "Logfile" 60
4.3 Das Hauptprogramm und seine Argumente 62
4.4 Linker-Bibliotheken 63
4.5 Initialisierungen und das große Aufräumen 64
4.6 Mit und ohne Startup-Code 65
4.7 Der Startup-Code 67
4.8 Hunks 68
5.1 Was ist neu? 69
5.2 Das Konzept 70
5.3 Optionen 71
5.3.1 Allgemeines 71
5.3.2 Ergebnisse und Ausgaben 72
5.3.2.1 Assembler-Quelltext: -a und -as 72
5.3.2.2 Objektdateien: -c 73
5.3.2.3 Ausführbare Programmdatei: -o 73
5.3.3 Preprozessor 73
5.3.3.1 Definieren von Symbolen: -d 73
5.3.3.2 Includedateien: -i 74
5.3.4 Compiler-Modus 74
5.3.4.1 ANSI C oder C++: -pc und -pp 74
5.3.4.2 Exception Handling: -px, -pX und -X 74
5.3.4.3 Erzeugen von Template-Funktionen: -t und -T 75
5.3.5 Codeerzeugung 75

5.3.5.1 Breakpoints setzen: -gb 75
5.3.5.2 Datenmodell: -gd und -gD 75
5.3.5.3 Zusammenfassen von Hunks: -gh 75
5.3.5.4 Ganzzahlige Arithmetik: -gm 75
5.3.5.5 Optimierungen: -go und -gO 76
5.3.5.6 Code für besondere Prozessoren: -g00, -g20, -g80 und -g81 76
5.3.5.6 Linken ohne Startup-Code: -gs, -gsi und -gsw 76
5.3.5.7 Code für Link-Libraries: -gl, -gL und -L 77
5.3.6 Debugger 77
5.3.6.1 Maxon Debugger: -b, -bi und -b0 77
5.3.6.2 Symbolischer Debugger: -bs und -bs0 77
5.3.7 Warnungen und Fehler 77
5.3.7.1 Format der Fehlerausgabe: -e, -el und -ei 77
5.3.7.2 Farben und Stile: -eb, -eB und -ec 78
5.3.7.3 Fehlerdatei: -ef 78
5.3.7.4 Optionale Warnungen: -w 78
5.3.7.5 Warnungen wie Fehler behandeln: -ew und -eW 79
5.3.8 Linker 79
5.3.8.1 Library-Pfade: -j 79
5.3.8.2 Linker-Libraries: -l 79
5.3.9 Die integrierte Projektverwaltung: -m, -mx und -mX 80
5.3.10 Arbeitsspeicher: -w 80
5.3.11 Vorcompilierte Headerdateien: -H, -h und -h0 81
5.4 Voreinstellungen mit "CPPOPT" 81
5.5 Zusammenfassung 82
6.1 Installation von Edward auf der Festplatte 85
6.2 Aufruf von Edward 85
6.2.1 Start von der Workbench 85
6.1.2 Start aus dem CLI 86
6.2 Eingabe und Bearbeitung von Texten 86
6.2.1 Die Status-Anzeige 86
6.2.2 BACKSPACE und DELETE 87
6.2.3 UndoLine-Funktion 87
6.2.4. RETURN, ENTER und TAB 87
6.2.5 Die Cursor-Tasten und der Ziffernblock 88
6.2.6 Der Schieberegler 88
6.2.7 Die Maus 88
6.3 Text 89

6.4.1 Erzeugen von Texten 89
6.3.2 Wahl eines bestimmten Textes 89
6.3.3 Freigeben eines Textes 90
6.3.4 Löschen des aktuellen Textes 90
6.3.5 Drucken eines Textes 90
6.4 Dateioperationen 91
6.4.1 Speichern von Texten 91
6.4.2 Laden von Texten 92
6.5. Blockoperationen 92
6.5.1 Cut, Copy, Paste 92
6.5.2 Spaltenblöcke 94
6.5.3 Laden, Speichern und Drucken 94
6.5.4 Einrücken 95
6.5.5 Umwandeln in Groß- bzw. Kleinbuchstaben 95
6.6 Suchen und Ersetzen 96
6.6.1 Das „Find"-Formular 96
6.6.2. Das „Find & Replace"-Formular 97
6.6.3 Struktur-Check (Klammer-Check) 98
6.6.4 Compiler Error-Datei interpretieren, Fehler anspringen 99
6.6.5 Auf bestimmte Zeilen positionieren 99
6.7 Strukturieren von Texten, Lesezeichen 100
6.7.1 Falten 100
6.7.2 Auszeichnen von Zeilen 102
6.7.3 Cursor-Bewegung 102
6.7.4 Lesezeichen (Bookmarks) 102
6.9 „Word-Wrap" 104
6.10 Makros 105
6.10.1 Makros aufzeichnen & abspielen 105
6.10.2 Makros speichern & abspielen 106
6.11 Fenster 107
6.11.1 Erzeugen von Fenstern 107
6.11.2 Wahl eines bestimmten Fensters 107
6.11.3 Schließen von Fenstern 108
6.11.4 Arbeiten mit mehreren Ansichten eines Textes 109
6.12 Verlassen des Editors 110
6.12.1. Unterbrechen der Arbeit 110
6.12.2 Verlassen des Editors 111

6.13 Iconify & HotKey 111
6.13.1 Iconify 111
6.13.2 HotKey 111
6.14 Der Kommandozeilen-Interpreter 112
6.15 Globale Einstellungen 112
6.15.1 Das „Globale Einstellungen"-Formular 112
6.15.2 Das „Seitenformat"-Formular 116
6.16 Benutzerdefinierte Menüs, Tastenbelegung, HotKey 117
6.16.1 Kurze Einführung in die EBNF-Darstellung 118
6.16.2 Definition eigener Menüs 119
6.16.3 Definition der Tastaturbelegung 120
6.16.4 Definition des HotKeys 123
6.16.5 Einladen einer neuen Definition 123
6.17 ARexx - Kommunikation mit Edward 123
6.18 Zurücknehmen von Text-Änderungen 124

Die Kommandos des Editors 126

1 Eingabe und Bearbeitung von Texten 127
2 Verwalten von Texten 131
3 Dateioperationen 133
4 Blockoperationen 134
5 Suchen und Ersetzen 137
6 Strukturieren von Texten 139
7 Ändern von Optionen für den aktuellen Text 142
8 Word-Wrap 143
9 Makros 144
10 Fenster 145
11 Verlassen des Editors 146
12 Formulare 147
13 Der Kommandozeilen-Interpreter 148
14 Ändern von globalen Einstellungen 149
15 ARexx - Kommunikation mit Edward 151
15.1 Abfrage von Statuswerten Edwards : 151

15.2 Laden von Texten aus Edward heraus 154
16 UnDo/ReDo - Zurücknehmen von Textänderungen 155
17 Verschiedenes 156

Anhang A 157
Veränderungen seit der Version 1.0 157
A-1. Compiler 157
A-1.1 Preprozessor 157
A-1.2 Parser und Sprachstandard 157
A-1.3 Codeerzeugung 158
A-1.4 Linker 158
A-2. Bibliotheken 159
A-3. Sonstiges 159
B-1. Warnungen 160
B-2. Fehlermeldungen 162
B-3. Sonstige Fehlermeldungen 185

Anhang-C – Editor-Fehlermeldungen 187
C-1 Informationen 187
C-2 Allgemeine Fehlermeldungen 187
C-3 Fehlermeldungen mit Abfragen 191
C-4 Meldungen mit Abfragen 191
C-5 Fehler bei Betriebssystemoperationen 193
C-6 Eingabefehler 195
C-7 Fehler in den Definitionsdateien 196
C-8 Fehler beim Analysieren von Kommandos 197
C-9 Fehlermeldungen beim Auswerten von Kommandos 199

Tutorial

Einführung in C++ 203
Ein paar kurze Bemerkungen, bevor es los geht 203

1. Der Einstieg 204
1.1 Ein erstes Beispiel 204
1.2 Lexikalisches 206
1.3 Zahlen, Variablen und Berechnungen 208
1.4 Numerische Datentypen und Operatoren 210
1.4.1 Ganzzahlige Datentypen 210
1.4.2 Literale für ganze Zahlen und Zeichenketten 212
1.4.3 Ganzzahlige Operatoren und Berechnungen 214
1.4.3.1 Allgemeines 214
1.4.3.2 Postfix-Operatoren 215
1.4.3.3 Unäre Operatoren 215
1.4.3.4 Multiplikative Operatoren 217
1.4.3.5 Addition und Subtraktion 218
1.4.3.6 Shift-Operatoren>> 218
1.4.3.7 Vergleiche 220
1.4.3.8 Bitweise Verknüpfungen 221
1.4.3.9 Logische Operatoren 221
1.4.3.10 Der bedingte Ausdruck 222
1.4.3.11 Zuweisungen 223
1.4.3.12 Der Komma-Operator 224
1.4.4 Fließkommatypen 224
1.4.4.1 Literale und Typen 224
1.4.4.2 Interne Darstellung der Daten 225
1.4.4.2 Fließkommaoperationen 226
1.4.5 Auswertung von Ausdrücken 226
1.4.6 Explizite Typumwandlungen 228
1.4.7 Initialisierungen 229
1.5 Anweisungen 230
1.5.1 Bedingungen und Verzweigungen 230
1.5.2 Anweisungen, Deklarationen und Blöcke 232
1.5.3 Schleifen 235
1.5.3.1 Die „while"-Anweisung 235
1.5.3.2 Die „for"-Schleife 236

1.5.3.3 Die „do"-Schleife 237
1.5.4 Sprünge 238
1.5.4.1 Sinn und Unsinn von „goto" 238
1.5.4.2 „break" und „continue" 240
1.5.5 Vielfachverzweigungen 242
1.5.6 Die „return"-Anweisung 244
1.6 Funktionen 244
1.6.1 „main" allein genügt nicht 244
1.6.2 Parameter und Argumente 245
1.6.3 Rückgabe von Werten 247
1.6.4 Funktionen und Gültigkeitsbereiche 249
1.6.4.1 Deklarationen auf Dateiebene 249
1.6.4.2 Rekursive Funktionen 251
1.6.4.3 Deklarationen und Definitionen 251
1.6.5 Überladen, Default-Parameter und mehr 253
1.6.5.1 Überladen von Funktionen 253
1.6.5.2 Abkürzen durch Default-Argumente 255
1.6.5.3 Variable Parameterlisten 257
1.6.6 Anachronismen aus der Mottenkiste 258
1.6.6.1 Umschalten in den C-Modus 258
1.6.6.2 Parameterlisten im alten Stil 259
1.6.6.3 Von der Steinzeit in die ANSI-Ära 260
1.6.7 Einige Standardfunktionen 261
1.6.7.1 Allgemeines 261
1.6.7.2 Ein- und Ausgabe: <stdio.h> 262
1.6.7.3 Zeichen, Zeichenketten und Bytefolgen: 262
1.6.7.4 Fließkommafunktionen: <math.h> 263
1.6.7.5 Die C-Fundgrube: <stdlib.h> 263

2. Datentypen und Deklarationen 265
2.1 Vektoren, Zeiger und Referenzen 265
2.1.1 Pointer 265
2.1.1.1 Von Objekten und Adressen 265
2.1.1.2 Zeiger: Deklaration und elementare Operationen 265
2.1.1.3 Zeiger als Funktionsparameter 267
2.1.1.4 Zeigertypen 268
2.1.1.5 Was man sonst noch so mit Zeigern machen kann 270
2.1.2 Vektoren 271
2.1.2.1 Das Vektor-Konzept in C 271
2.1.2.2 Zeiger und Vektoren 272

2.1.2.3 Genaueres über Vektordeklarationen 275
2.1.2.4 Vektoren als Parameter 277
2.1.2.5 Strings 278
2.1.2.6 Initialisierung von Vektoren 286
2.1.3 Referenzen 288
2.1.3.1 Einführung 288
2.1.3.2 „VAR-Parameter" und temporäre Objekte 289
2.1.3.3 Weitere Möglichkeiten und Einschränkungen 291
2.2 Qualifizierungen, Deklarationen und Spezifikationen 292
2.2.1 Speicherklassen 292
2.2.1.1 Automatische und statische Daten 292
2.2.1.2 Module und ihre Gültigkeitsbereiche 294
2.2.1.3 Daten in Registern 301
2.2.1.4 Daten im CHIP-RAM 303
2.2.2 Typdefinitionen 304
2.2.3 Konstante Daten auf der Flucht 305
2.2.3.1 Die „const"-Qualifizierung 305
2.2.3.2 Flüchtige Daten: „volatile" 308
2.2.4 Inline-Funktionen 309
2.3 Zeiger auf Funktionen 311
2.3.1 Wohin zeigen Zeiger auf Funktionen? 311
2.3.2 Anmerkungen zu Syntax und anderen Details 312
2.3.3 Zeiger auf Überladene Funktionen 313
2.3.4 Eine Anwendung: „qsort" und „bsearch" 314
2.4 Aufzählungen 316
2.4.1 Deklaration 316
2.4.2 Typnamen und Typlabel 318
2.4.3 Aufzählungen und Zahlen 319
2.5 Strukturen 321
2.5.1 Grundlagen 321
2.5.2 Definitionen und Deklarationen 323
2.5.3 Initialisierung 325
2.5.4 Gültigkeitsbereiche und Scope-Regeln 326
2.5.5 Unionen 328
2.5.6 Bitfelder 332
2.6 Dynamisch organisierte Datenstrukturen 333
2.6.1 Die Idee 333
2.6.2 „new" und „delete" 334

2.6.3 Dynamisch erzeugte Vektoren 335
2.6.4 Initialisierungen 336
2.6.5 „malloc" und andere Antiquitäten 337
2.6.6 Beispielprogramm: Eine Liste von Daten 339
2.6.6.1 Das Problem: Ein wirrer Haufen von Daten 339
2.6.6.2 Die Lösung: Eine doppelt verkettete Liste 341
2.6.6.3 Initialisieren und Einfügen 343
2.6.6.4 Suchen und Löschen 346
2.6.6.5 Das Programm an einem Stück 348

3. Objektorientiertes Programmieren 353

3.1 Klassen und Objekte 353
3.1.1 Keine Panik! 353
3.1.2 Member-Funktionen 355
3.1.2.1 Erste Beispiele 355
3.14.2.2 Member-Funktionen (Methoden) und konstante Objekte 357
3.1.2.3 Inlining von Member-Funktionen 357
3.1.3 Klassen und Zugriffsrechte 358
3.1.4 No Scope, no hope! 360
3.1.5 Friends 362
3.1.6 Statische Member 364

3.2 Vererbungslehre 366
3.2.1 Abgeleitete Klassen 366
3.2.1.1 Ein erstes Beispiel 366
3.2.1.2 Mehrfach-Vererbung und andere Features 369
3.2.1.3 Vererbung und Typregeln 371
3.2.2 Scoperegeln und Zugriffskontrolle bei Vererbung 373
3.2.2.1 Noch mehr Details 373
3.2.2.2 Der Zugriffsmodus „protected" 374
3.2.2.3 Zugriffs-Deklarationen 375
3.2.3 Virtuelle Member-Funktionen 377
3.2.3.1 Ein Beispiel 377
3.2.3.2 Die Details 379
3.2.3.3 Abstrakte Klassen 381
3.2.4 Ein paar Anmerkungen zur Implementation 382
3.2.5 Virtuelle Basisklassen 385

3.3 Konstruktoren und Destruktoren 393
3.3.1 Initialisierung von Klassen 393
3.3.2 Konstruktoren mit Argumenten 394

3.3.3 Initialisierung von Basisklassen und Membern 396
3.3.4 Besondere Konstruktoren 400
3.3.4.1 Default-Konstruktoren 400
3.3.4.2 Kopier-Konstruktoren 401
3.3.5 Destruktoren 403
3.4 Beispielprogramm: 406
3.5 Noch mehr Features 420
3.5.1 Zeiger auf Member 420
3.5.2 Konvertierungsfuntionen 423
3.5.3 Temporäre Objekte 426
4. Überladen 429
4.1 Überladene Funktionen 429
4.1.1 Was man darf, und was nicht 429
4.1.2 Was der Compiler davon hält 430
4.1.2.1 Die Hierarchie der Typwandlungen 430
4.1.2.2 Matching von Funktionsargumenten 435
4.2 Operatoren 437
4.2.1 Allgemeine Operatoren 437
4.2.1.1 Allgemeines über allgemeine Operatoren 437
4.2.1.2 Wie man so etwas deklariert 439
4.2.1.3 Und wie man es benutzt 440
4.2.2 Spezielle Operatoren 441
4.2.2.1 Der Zuweisungs-Operator „=“ 441
4.2.2.2 „new“ und „delete“ 443
4.2.2.3 Funktionsaufruf als Operator 450
4.2.2.4 Klassen als Möchtegern-Vektoren 451
4.2.2.5 Intelligente Zeiger: "->" und das unäre ""* 453
4.2.2.6 Inkrement und Dekrement 454
4.2.2.7 Ein- und Ausgabe 456
4.2.3 Beispielprogramm: Die „string“-Bibliothek 457
5. Der Preprozessor 465
5.1 Allgemeines 465
5.2 Was Sie schon immer über #define wissen wollten 466
5.2.1 Makrodefinition 466
5.2.2 Makros mit Parametern 469
5.3 Was Sie noch nie über #include wissen wollten 470

5.4 Bedingte Übersetzung 471
5.5 Pragmatismus 473
5.5.1 Allgemeines und Wiederholungen 473
5.5.2 Breakpoint-Kontrolle 474
5.5.3 Amiga-spezifische Systemaufrufe 475
5.6 Noch mehr Features 476
5.6.1 Verbinden von Zeilen 477
5.6.2 Verbinden konstanter Zeichenketten 477
5.6.3 Trigraph-Sequenzen 477
5.6.4 Hausgemachte Fehlermeldungen 479
5.6.5 Getürkte Dateinamen und Zeilennummern 479
5.6.6 Nix dahinter 480
5.7 Vordefinierte Symbole 480
5.8 Arbeitsweise des Preprozessors 482

6. Was Sie als Amiga-Entwickler wissen sollten 485
6.1 Die Schnittstelle zum Betriebssystem 485
6.2 Start von der Workbench 485
6.2.1 Die Do-It-Yourself-Methode: die Funktion "wbmain" 485
6.2.2 Die einfache Methode: die Datei "wbstartup.h" 488

7. Templates 491
7.1 Einführung 491
7.2 Klassen-Templates 491
7.2.1 Ein einfaches Beispiel 491
7.2.2 Deklarationen und Definitionen 493
7.2.3 Member-Funktionen 495
7.2.4 Statische Member 496
7.2.5 Friends 497
7.3 Funktions-Templates 498
7.3.1 Beispiel: Sortieren von Vektoren 498
7.3.2 Deklaration und Definition 499
7.3.3 Handgestrickt kontra compilergeneriert 499
7.3.4 Argumemt-Matching beim Aufruf von Templatefunktionen 501
7.3.4.1 Benutzung von Funktionstemplates 501
7.3.4.2 Mischung von Funktiontemplates und "echten" Funktionen 502

7.3.5 Generierung von Templatefunktionen 503
7.4 Beispielprogramm: Schon wieder verkettete Listen 504

8. Ausnahmebehandlung 509
8.1 Worum gehts? 509
8.2 Vom Werfen und Auffangen 510
8.3 Von Ausnahmen und ihren Typen 512
8.4 Vom Fangen und wieder Wegwerfen 515
8.5 Von Konstruktoren und Destruktoren 517
8.6 Von erwarteten und unerwarteten Ausnahmen 520
8.7 Beispielprogramm: Das Übliche 523

Referenz 525

1. Die ANSI C Includedateien 527
1.1 Verschiedene nützliche Funktionen: <stdlib.h> 527
1.1.1 Umwandlung zwischen Zahlen und Strings 527
1.1.2 Mathematische Funktionen 529
1.1.3 Zufallszahlen 531
1.1.4 Speicherverwaltung, Typen und Zeiger 532
1.1.5 Programmende 534
1.1.6 Kommunikation mit dem Betriebssystem 535
1.1.7 Sortieren und Suchen 535
1.2 Ein- und Ausgaben: <stdio.h> 536
1.2.1 Typen, Objekte und Konstanten 536
1.2.2 Öffnen und Schließen von Dateien 538
1.2.3 Strings und Zeichenketten 540
1.2.4 Formatierte Ausgabe 542
1.2.5 Formatierte Eingabe 546
1.2.6 Dateioperationen 549
1.2.6.1 Dateiende und Fehlerbehandlung 549
1.2.6.2 Puffern 550
1.2.6.3 Externe Dateien 552
1.2.6.4 Binäre Daten 553
1.2.6.5 Positionieren in Dateien 554
1.2.7 Sonstige Funktionen 555

1.3 Fehlersuche: <assert.h> 556
1.4 Zeichenklassen: <ctype.h> 557
1.5 Fehlernummern: <errno.h> 559
1.6 Ganzzahlige Grenzwerte: <limits.h> 561
1.7 Mathematische Funktionen: <math.h> 562
1.7.1 Umwandlung von Zahlen in Zeichenketten 562
1.7.2 Fließkomma-Berechnungen 564
1.8 Haarsträubende Sprünge: <setjmp.h> 567
1.9 Signale und Ereignisse: <signal.h> 570
1.10 Variable Argumentlisten: <stdarg.h> 572
1.11 Implementationsabhängige Definitionen: <stddef.h> 575
1.12 Zeichenketten und Speicherverwaltung: <string.h> 576
1.13 Datum und Uhrzeit: <time.h> 582

2. Bibliotheken für C++ 587
2.1 Ein- und Ausgaben in C++: <stream.h> und <iostream.h> 587
2.1.1 Ausgabe 587
2.1.1.1 „ostream" und der Operator „«" 587
2.1.1.2 Basen und Manipulatoren 589
2.1.1.3 Formatierte Ausgabe 591
2.1.1.4 Ausgabe binärer Daten 592
2.1.2 Eingabe 593
2.1.2.1 Texteingaben mit „»" 593
2.1.2.2 Eingabefunktionen auf „istream" 594
2.1.3 Weitere Funktionen 596
2.1.3.1 Die Schnittstelle zum ANSI C-Dateisystem 596
2.1.3.2 Fehlererkennung 596
2.2 Ströme und Dateien: <fstream.h> 597
2.2.1 Klassen für Dateien 597
2.2.2 Dateien öffnen und schließen 598
2.2.3 Fehlererkennung 599
2.2.4 Konstruktoren und Destruktoren 600
2.2.5 Puffer 601
2.2.6 Positionieren in Strömen 601

2.3 Ein paar wichtige Definitionen in <new.h> 602
2.4 Ausnahmebehandlung: <exception.h> 605
2.5 Eine String-Library: <tools/str.h> 606
2.5.1 Die Klasse „String" 606
2.5.2 Member-Funktionen 607
2.5.3 Der Operator „+" auf Strings 609
2.5.4 Vergleiche 610
2.5.5 Ein- und Ausgabe 611
2.6 Interna von MaxonC++: <streamdefs.h> 612
2.6 Funktionen für originelle Linker-Opt.: <linkerfunc.h> 615

Easy-Objects 617
Einführung 619
Die Ausnahmebehandlung 623
Container und Cursor 623
Das Handlerkonzept 625
Die Fensterklasse WindowC 627
Die Fensterklasse GTWindowC 630
Gadgets und Gadgetlisten 630
Der Layouter 632
Die Regeln zur Bescheibung eines Layouts 633
Der Layoutvorgang 639
Eine Knopfzeile 639
Eine Gadgetspalte 641
Die Grenzen des Layouters 643
Zur Gestaltung der Oberfläche 643
Die Fensterklasse LayouterWindowC 644
Die Fensterklasse StandardWindowC 645
Datei- und Verzeichnissperren 646
Die Dateiklassen 647

Kommandozeilenargumente 648
Beschreibung des HotHelp Projekts zu EASY-OBJECTS 648
Einführung 648
Hintergrund 648
Technische Kapitel 649
Benutzung des HotHelp Projekts EASY-OBJECTS 651

Source Level Debugger 655

1. Einführung 657
Was können Sie mit MaxonSDB tun? 657
Über dieses Handbuch 657

2. Die Benutzung von MaxonSDB 659
Die Vorbereitung Ihres Programms 659
Programmstart 659
Der Workbenchstart 659
Der Shellstart 660
Der Start aus der Entwicklungsumgebung 661

3. Die Fenster des Debuggers 662
Drag&Drop 662
Das Projektfenster 662
Das Quelltextfenster 663
Breakpoints 664
Das Modulfenster 664
Das Fenster der globalen Variablen 665
Den Knopf Inspiziere. 665
Das Fenster der Funktionen 666
Das Fenster der lokalen Variablen 667
Das Fenster der aktuellen Variablen 667
Das Fenster zur Überwachung 667
Die Konfiguration der Oberfläche 668

Das Menü Projekt 669
Das Menü Schritt 669
Das Menü Fenster 670
Das Menü Einstellungen 670

Anhang 671
Der Enforcer 671
Der SegTracker 671
inline-Funktionen 671
Technische Einzelheiten: 672
Forbid() und Disable() 672
Die Prozessoren 68020,68030 und 68040 673
Sicherheitsabfragen 674
Fehlermeldungen 675

HotHelp 3 677

1. Über diese Anleitung 679

2. Jederzeit hilfsbereit 680

3. Eigenschaften von HotHelp 681

4. Anforderungen 683

5. Installation 683
5.1. Installation auf Festplatte 683
5.2. Installation auf Disketten 684
5.3. Der Installationsvorgang 685
5.4. Installation von Hand 686
5.5. Installation mit englischer Benutzerführung 687

6. Die Bedienung von HotHelp 688
6.1. Die Titelzeile des HotHelp-Fensters 689
6.2. Rollsymbole 690
6.3. Querverweise 690
6.4. Das Vortext-Symbol 691
6.5. Blättern im Projekt 691
6.6. Schlüssel, Projekte und Muster 691
6.7. Abrufen der Projektübersicht 692
6.8. Die Kapitelübersicht 693
6.9. Der Export von Hilfstexten 693
6.10. Verschiedene Ansichten über HotHelp 694
6.11. Die integrierte Suchfunktion 695
6.12. Verwendung von Lesezeichen 696
6.13. Hilfe über Hilfe 696
6.14. Tastaturbedienung 697
6.15. Möglichkeiten, HotHelp aufzurufen 697

7. Onlineprojekte 699
7.1. Benutzung der Onlineprojekte 699
7.2. Anwendungsgebiete für Onlineprojekte 700
7.3. Onlineprojekte und andere Editoren 700

8. ARexx 701

9. Änderungen seit Version 2 702

10. Die HotHelpTools 704
10.1. Die Hilfsfunktion der HotHelpTools 704
10.2. Start der HotHelpTools 705
10.3. HotHelpPref - HotHelps Voreinstellungen 706
10.3.1. Fenster 1: Farben 707

10.3.2. Fenster 2: Zeichensatz 708
10.3.3. Fenster 3: Ausgabeformate 709
10.3.4. Fenster 4: Steuerung 710
10.3.5. Fenster 5: Hotkeys 711
10.3.6. Fenster 6: Online-Optionen 711
10.3.7. Fenster 7: HKH-Optionen 712
10.4. HotHelpPro - Der Projektmanager 714
10.4.1. Standardprojekte 714
10.4.2. Onlineprojekte 715
10.4.3. Projekttest 716
10.4.4. AmigaGuide-Projekte 716
10.5. HotHelpMarker - der Projekte-Editor 718
10.5.1. Grundlagen der Projekterstellung 718
10.5.2. Hintergrund 719
10.5.3. Eingabe der Texte 719
Die Quelldateien 720
Die Texte 720
10.5.4. Markieren der Texte 720
Der HotHelp-Textmarker 720
Aufteilung des Textes 721
Einteilung in Einträge 721
Die Schlüssel 722
Querverweise 722
Grafiken 723
Schriftarten 723
Die Übersetzung 723
10.5.5. Resümee 724
10.5.6. Weitere Gestaltungsmöglichkeiten 724
Titel 725
Kapitel, Vorgänger und Nachfolger 725
Projekt- und ARexx-Querverweise 725
Schlüsselgruppen 725
10.6. EasyHotHelp 727
10.7. HotHelpComp - der HotHelp-Übersetzer 728
11. Letzte Worte 729

MaxonASM 731

1. Einleitung 733

2. Grundlagen der Assemblerprogrammierung 734

2.1 Warum Assembler? 734

2.2 Der Aufbau eines Quelltextes 734

3. Der Assembler 735

3.1 Die Adressierungsarten des MC 68000 735

3.2 Symbole und mathematische Ausdrücke 737

3.2.1 Globale Symbole 737
3.2.2 Lokale Symbole 738
3.2.3 Label 738
3.2.4 Variablen 738
3.2.5 Reservierte Symbole 739
3.2.5.1 NARG 739
3.2.5.2 __RS 739
3.2.5.3 _MOVEMBYTES 739
3.2.6 Mathematische Ausdrücke 740
3.2.7 Zahlenformate 740
3.2.8 Operationen und deren Prioritäten 740

3.3 Macros 741

Schlüsselwörter/Funktionen 743
1. *VALOF(<Symbol>) 743
2. *STRLEN(<String>) 744
3. *LEFT(<String>,n) 744
4. *RIGHT(<String>,n) 744
5. *MID(<String>,s,n) 744
6. *UPPER(<String>) 745

3.4 Bedingte Assemblierung 746

3.5 ALIGNment 747

3.6 Sektionierung 748

Direktiven zur Steuerung der Code-Erzeugung 749

3.7 Erzeugung von absolutem Code 750

3.8 Konstanten 751
3.9 Reservierte Speicherblöcke 753
3.10 Definition einer Variablen 753
3.11 Definition von Strukturoffsets 754
3.12 Externe Quelltexte 754
3.13 Externe Rohdateien 755
3.14 Copper-Assembler 755
3.15 Externe Symbole 758
3.16 Codearten 760
3.17 Assemblerbetriebsarten 760
3.18 Mögliche Optimierungen 761
3.19 Ausgabedateien 763

Anhang A 765
Die Befehle des MC 68000 / MC 68010 765
Die zusätzlichen Befehle des MC 68010 768

Anhang B 769
Die Assemblerdirektiven 769

Anhang C: Die Fehlermeldungen 773
DOS-Fehler beim Dateihandling 773
Fehler in der Sektionierung 773
MACRO-Fehler 773
Fehler beim Parsen einer Zeile 775
Fehler bei der Labeldefinition 775
Fehler im math. Ausdruck 775
Fehler im Addressmode 776
Fehler im ersten Befehlsoperanden 777
Fehler im zweiten Befehls-Operanden 778

Fehler im dritten Befehls-Operanden 778
Fehler in den IF-Konstruktionen 778
Spezielle Fehler bei Branch-Befehlen 779
Spezielle Fehler bei EQU 779
Spezielle Fehler bei EQUR 779
Spezielle Fehler bei REG 779
Spezielle Fehler bei SET 780
Spezielle Fehler bei CNOP/ALIGN 780
Spezielle Fehler bei XDEF 780
Spezielle Fehler bei XREF 780
Spezielle Fehler bei DC.x 780
Spezielle Fehler bei RS.x 780
Spezielle Fehler bei INCBIN 781
Spezielle Fehler bei BASEREG 781
Allgemeine Fehler 781
Interne Assembler-Fehler 781
Fehler bei FPU-Operationen 782
Fehler im Addressmode 782
Fehler im zweiten Befehlsoperanden 782
Fehler im Addressmode 782
Fehler im ersten Operanden 783
Fehler im zweiten Operanden 783
Fehler bei Copper-Operationen 783

Anhang D: Register 784

Anhang E, Befehle 785

AMIGA

MaxonC++

Anwenderhandbuch

MAXON computer

Maxon C++ 3

1.1 Was gibt es Neues?

Das MaxonC++ Compiler-System hat eine ganze Menge neuer Features bekommen. Das wichtigste ist sicher die Kompatiblität zum 3.0-Standard, weshalb auch die Versionsnummer 2.0 übergangen wurde. Der "3.0-Standard" wird wohl weitgehend mit dem lange erwarteten und immer noch nicht ganz fertigen ANSI C++-Standard identisch sein. Die Neuerungen gegenüber dem "2.0-Standard" (dem auch Maxon C++ 1.x folgte) sind zum einen die "Templates", durch die das Konzept der abstrakten Datentypen besser realisiert werden kann, und zum anderen die Ausnahmebehandlung, die die Zuverlässigkeit von Programmen zu steigern hilft.

Neben vielen anderen Verbesserungen bietet der MaxonC++ Compiler jetzt auch die Möglichkeit, speziellen Code für 32Bit-Prozessoren und FPUs erzeugen zu lassen. Schon seit der Version 1.1 ist es möglich, Includedateien "vorcompilieren" zu lassen. Das beschränkte sich aber auf ein Umkopieren der Dateien in Token-Form, was keine besonders dramatische Beschleunigung des Übersetzungsvorgangs brachte. Jetzt kann der Compiler, nachdem er die Header-Dateien gelesen hat, seinen gesamten internen Zustand abspeichern und beim nächsten Übersetzungsvorgang genau dort wieder einsetzen. Dadurch schrumpfen die Compile-Zeiten für Programme, die viele Includes benutzen, drastisch zusammen.

Die integrierte Umgebung MCPP wurde komplett überarbeitet. Einmal davon abgesehen, daß alles viel schöner und bequemer wurde, ist das erheblich verbesserte integrierte „Make" besonders hervorzuheben. Viele dieser Verbesserungen kamen auch dem „Make" im MCPPC zugute. Der Maxon Assembler (nur in der Developer-Version enthalten) wurde besser in die Umgebung integriert, indem MCPP die Fehlermeldungs-Dateien des Assemblers liest und im Compiler-Fehlermeldungswindow anzeigt. Außerdem besitzt MCPP jetzt einen eigenen ARexx-Port.

Auch beim "Zubehör" hat sich einiges getan: Es gibt einen komplett neugeschriebenen und in vieler Hinsicht verbesserten Debugger sowie eine Klassenbibliothek, die nicht nur eine objektorientierte Schnittstelle zu den wichtigsten Bereichen des Amiga-Betriebssystems darstellt, sondern viele alltägliche Programmieraufgaben erheblich vereinfacht.

Die einzige Änderung, die zu Inkompatiblitäten mit der ersten Version führen kann, ist die Umstellung des Datentyps "float" vom "FFP-Single"- auf das "IEEE-Single"-Format. Dadurch erhält "float" die Genauigkeit und den Wertebereich, den der ANSI-C-Standard fordert.

1.2 Der Umstieg

1.2.1 Konfigurationsdateien

Ihre alte Konfigurationsdatei „*S:MaxonC++.config*" können Sie leider nicht mehr verwenden.Wenn Sie den alten Compiler nicht mehr verwenden, können Sie diese Datei löschen. Die neue Konfiguration heißt „*S:mcpp.config*" und hat ein vollständig anderes Format. Dafür gibt es jetzt die Möglichkeit, alle Einstellungen einschließlich Größe und Lage der verschiedenen Fenster in einer Datei „*mcpp.SESSION*" abzuspeichern. Das funktioniert also ganz analog zum „*EDWARD.SESSION*" des Editors.

1.2.2 „Edward.DEF"

Wenn Sie bereits eine ihren speziellen Bedürfnissen angepaßte Editor-Konfigurationsdatei "*Edward.DEF*" besitzen, werden Sie diese wohl weiter verwenden wollen. Das ist ohne weiteres möglich, allerdings sind beim *MCPP* viele Features neu hinzugekommen und dafür überflüssige entfernt worden, so daß Sie sinnvollerweise Ihre Menü- und Tastenbelegung überarbeiten sollten. Dabei werden Sie zwei neue Edward-Features nützlich finden: Zum einen kann in der "*Edward.DEF*"-Datei mit Steuerzeilen wie `"#ifdef MCPP1"`, `"#ifdef MCPP3"` oder `"#ifdef MPASCAL3"` zwischen den verschiedenen Applikationen, in die Edward inzwischen eingebunden wird, unterschieden werden (bisher gab es ja nur die allgemeine Modusabfrage mit `"#ifdef COMPILER"`); zum anderen können in "*Edward.DEF*" jetzt Include-Dateien verwendet werden. Da die Edward-Konfiguration so auf verschiedene Dateien verteilt wird, ist sie bei künftigen Updates leichter zu pflegen.

Wichtig

Bei der Installation des neuen Compilers wird eine eventuell von einer alten Version von EDWARD vorhandene „EDWARD.DEF"-Datei umbenannt und die neue installiert.

Sollten Sie modifikationen an Ihrer alten „EDWARD.DEF"-Datei vorgenommen haben, müssen Sie diese in die neue übertragen.

Die Abkürzungen für MCPP-Kommandos werden wegen Zweifels an deren Nützlichkeit generell nicht mehr unterstützt. Es muß jetzt also z. B. `SendAppComm "compile"` statt `SendAppComm "c"` heißen.

Alle alten und neuen Funktionen der Projektverwaltung sind jetzt über einen ganz neuen Dialog erreichbar. Dadurch kann das überladene, unübersichtliche und umständliche **„Make"-Menü** komplett entfallen. Auch sonst wurden einige Anweisungen überflüssige oder zumindest wenig nützliche Anweisungen weggelassen. Die Kommandos:

`addfiletx, addobj, mkinstr, mkdep, remobj, remtx, touch, opentext, chdir und compfile`

werden nicht mehr unterstützt, während die Make-Anweisungen

```
newmk, loadmk, addedittx, addtxobj und leavemk
```

prinzipiell weiter benutzt werden können. Am besten ist es aber, das **„Make"-Menü** durch einen schlichten Menüpunkt wie

```
ITEM "Projektverwaltung" (SendAppComm "make") "M"
```

im **„Compiler"-Menü** zu ersetzen. Diese Funktion öffnet den erwähnten Make-Requester - oder den Filerequester, falls noch keine Projektdatei geladen ist. Im letzteren Fall können Sie sowohl eine bestehende Projektdatei auswählen als auch den Namen einer neu zu erzeugenden Datei angeben.

Eine andere sinnvolle Ergänzung wäre ein Menüpunkt

```
ITEM "Fehlermeldungen" (SendAppComm "errors")
```

(natürlich können sie genausogut auch eine Taste mit dieser Funktion belegen).

Damit ist es jetzt möglich, das bereits geschlossene Compiler-Window erneut zu öffnen, z. B. um einen erneuten Blick auf die letzten Fehlermeldungen zu werfen. `SendAppComm "saveconf"` öffnet in der aktuellen Version keinen Filerequester mehr, sondern schreibt die Konfiguration grundsätzlich nach *„S:mcpp.config"*. Übrigens enthält diese Datei jetzt endlich wirklich **alles**, was es über die Umgebung und ihre Einstellungen zu sagen gibt. Neu ist `SendAppComm "savesession"` Diese Anweisung schreibt die Konfiguration in eine Datei *„mcpp.SESSION"* im aktuellen Verzeichnis, von wo sie beim nächsten Programmstart aus dem gleichen Verzeichnis automatisch geladen wird. Und weil dies analog zur Session-Datei des Edward ist, sollte überall, wo in *„Edward.DEF"* das *„SaveSession"* des Editors aufgerufen wird, das *MCPP-"SaveSession"* ergänzt werden. Aus einem Menüpunkt

```
ITEM "Projekt speichern"  (SaveSession)
```

könnten Sie also

```
ITEM "Projekt speichern" ((SaveSession)(SendAppComm "savesession"))
```

machen.

Eine Compiler-Menüdefinition könnte z. B. so aussehen:

```
TITLE "Project"  ... Normale Editor-Projekt-Menüs ...
ITEM "Save Session"  ((SaveSession) #ifdef COMPILER (SendAppComm
"savesession")#endif ) "U"
#ifdef MCPP3
TITLE "Compiler"
ITEM "übersetzen"   (Sendappcomm "compile")
ITEM "Debugger"     (SendAppComm "debug") "D"
EMPTY
ITEM "Ausführen"    (SendAppComm "run")
ITEM "Abbrechen"    (SendAppComm "break")
```

```
EMPTY
ITEM "Exe-Datei"    (SendAppComm "exefile")
ITEM "Objektdatei"  (SendAppComm "objectfile")
EMPTY
ITEM "Information"  (SendAppComm "info")
ITEM "Fehlermeldungen"  (SendAppComm "errors")
TITLE "Make"
ITEM "Neues Makefile"  (SendAppComm "newmk")
ITEM "Makefile laden"  (SendAppComm "loadmk")
ITEM "Projektverwaltung"  (SendAppComm "make") "M"
EMPTY
ITEM "Text aus Editor aufnehmen"  (SendAppComm "addedittx")
ITEM "Text und Objekt aufnehmen"  (SendAppComm "addtxobj")
EMPTY
ITEM "Make verlassen"  (SendAppComm "leavemk")
TITLE "Optionen"
ITEM "System"  (SendAppComm "opsys")
ITEM "Compiler"  (SendAppComm "opcomp")
ITEM "Linker"  (SendAppComm "oplink")
EMPTY
ITEM "Compiler-Optionen sichern"  (SendAppComm "saveconf")
ITEM "Compiler-Session sichern"(SendAppComm "savesession")  #endif
```

Das folgende wäre eine praktische Tastenbelegung:

```
#ifdef COMPILER
(
F9              (Sendappcomm "Run")
F10             (Sendappcomm "ExeFile")   )

(SHIFT
F9              (Sendappcomm "Compile")
F10             (Sendappcomm "ObjectFile")   )

(CTRL
F9              (Sendappcomm "Errors")
F10             (Sendappcomm "Info")   )
#endif
```

1.2.3 Dateien und Verzeichnisse

Alle ausführbare Programme liegen jetzt im Verzeichnis „*MCPP:bin*“, was in früheren Versionen nicht immer so einheitlich geregelt war. Genau dort sollte auch die Datei „*MCPP.RCT*“ liegen. Sie ist nötig, weil die neue Oberfläche von MCPP mit MakeAPP erzeugt wurde.

1.3 Ein ersterÜberblick

1.3.1 Disketten

MaxonC++ wird auf mehreren Disketten ohne Kopierschutz geliefert. Die genaue Zahl und den Inhalt der Disketten erfahren Sie, wenn Sie die *„ReadMe"-Datei* auf der ersten Diskette lesen. Da sich nach Drucklegung dieses Handbuchs noch Änderungen ergeben können, wählten wir diesen Weg.

1.3.2 Installation

Um das System auf Ihrer Festplatte zu installieren, sollten Sie sich zuerst ein geeignetes Verzeichnis anlegegen, z. B. *„DH0:MCPP"*. Sie sollten Dateinamen mit einem „+" am Ende meiden, da das CLI bzw. die Shell bei Eingaben, die mit „+" enden, Schwierigkeiten bereiten.

Nun starten Sie einfach das Installationsprogramm von der ersten Diskette.

1.3.3 Die Komponenten des MaxonC++ Systems

In diesem Abschnitt sollen die wichtigsten Dateien des MaxonC++ Systems kurz erläutert werden. Das soll Ihnen helfen, sich mit den zahlreichen Dateien und Verzeichnissen besser zurechtzufinden.

Bei der Installation wird ein logisches Gerät *„MCPP:"* dem Verzeichnis, in das Sie die Installation durchgeführt haben, zugewiesen. Darin befindet sich ein Unterverzeichnis *„Bin"*, in dem die meisten ausführbaren Dateien stehen.

"MCPP" ist die integrierte Entwicklungsumgebung. Wenn die Installation korrekt und erfolgreich abgelaufen ist, können Sie versuchen, dieses Programm zu starten.

MCPP enthält das komplette Entwicklungssystem einschließlich Compiler und Projektverwaltung - nur der Editor muß noch dazugeladen werden. Während der Installation wird der Editor Edward in ein Verzeichnis *„Edward:"* abgelegt, von wo MCPP ihn automatisch nachlädt. Sie können Edward aber auch allein ohne Compiler aufrufen.

Edward benötigt unbedingt eine Datei *„Edward:Edward.DEF"*, in der die Menü- und Tastenbelegung der Entwicklungsumgebung abgespeichert ist. Diese Datei wird bei der Installation autmatisch erzeugt. Im Edward-Handbuch finden Sie eine ausführliche Anleitung, wie Sie diese Datei ändern können, um das MaxonC++ nach Ihren persönlichen Vorstellungen und Wünschen zu konfigurieren.

Sowohl MCPP als auch Edward benötigen die *„rct.library"*. Diese sollte während der Installation in Ihr *„Libs:"*-Verzeichnis kopiert worden sein.

Neben einigen nützlichen anderen Funktionen erleichtert die *„rct.library"* insbesondere die Programmierung von Benutzeroberflächen, die mit dem Oberflächengenerator *„MakeAPP"* erzeugt wurden. Ein solches Programm benötigt dann aber nicht nur diese Library, sondern auch eine

„*.RCT*"-*Datei*, in der die eigentliche Oberfläche abgespeichert ist. Für MCPP heißt diese Datei „*MCPP.RCT*" und wird im Verzeichnis „*MCPP:Bin*" erwartet, während die Edward-Oberfläche in „*EdwardRes.RCT*" aus dem Verzeichnis „*Edward:*" abgespeichert ist.

Um MCPP zu starten, müssen also zusammengefaßt folgende Dateien installiert sein und im richtigen Verzeichnis liegen:

- *MCPP:Bin/MCPP*
- *MCPP:bin/MCPP.RCT*
- *EDWARD:Edward*
- *EDWARD:Edward.DEF*
- *EDWARD:MCPP3.DEF*
- *EDWARD:EdwardRes.RCT*
- *LIBS:rct.library*

In der Entwicklungsumgebung können Sie über Dialoge eine ganze Menge Einstellungen vornehmen. Damit Sie das nicht bei jedem Start des Systems neu tun müssen, können Sie die Einstellungen natürlich auch abspeichern, und zwar in zwei Dateien im Verzeichnis „*S:*":

- „*S:Edward.CONF*" für Editor-Einstellungen,
- „*S:MCPP.config*" für Compiler-Optionen.

Auch diese beiden Dateien werden bei der Installation automatisch angelegt. Wenn Sie fehlen, macht das aber nichts, da sowohl MCPP als auch Edward sinnvolle Default-Werte besitzen. Vorzugsweise hält das MaxonC++-System sich ohnehin nicht an die Dateien im „*S:*"-Verzeichnis, sondern versucht nach Möglichkeit, aus dem aktuellen Verzeichnis folgende Dateien zu lesen:

- „*Edward.SESSION*" für Editor-Einstellungen,
- „*mcpp.SESSION*" für Compiler-Optionen.

Diese beiden Dateien enthalten nicht nur sämtliche Einstellungen und Optionen, sondern auch Informationen darüber, welche Texte geladen und welche Fenster wo geöffnet sind. Die Session-Dateien werden aber nicht bei der Installation erzeugt, sondern auf Verlangen von *MCPP* bzw. Edward selbst ins aktuelle Verzeichnis geschrieben. Wenn eine dieser Dateien vorhanden ist, hat sie Vorrang vor der jeweils entsprechenden Konfigurationsdatei im Verzeichnis "S:". Durch die Session-Dateien können Sie den aktuellen Zustand am Ende einer "Sitzung" abspeichern und später wieder genau so wieder anfangen, wie Sie aufgehört haben.

"*mcppc*" ist die Kommandozeilenversion des Compilers. Falls Sie integrierte Entwicklungsumgebungen nicht mögen sollten, werden Sie wahrscheinlich dieses Programm von einer Shell aus benutzen. Naheliegenderweise liegt „*mcppc*" im Verzeichnis „*MCPP:Bin*".

Außerdem finden Sie im „*MCPP:bin*"-Verzeichnis (mindestens) folgende Programme:

- *MaxonASM:* Der Assembler (nur in Developer-Version)
- *MaxonSDB:* Der Source-Level-Debugger (nur in Developer-Version)

Ferner sollte nach erfolgreicher Installation auch irgendwo ein Verzeichnis namens *„HOTHELP:"* existieren, in dem alle Dateien und Verzeichnisse liegen, die zum Online-Hilfesystem *„HotHelp"* gehören.Um sinnvoll mit MaxonC++ arbeiten zu können, benötigen Sie auch Libraries und Include-Dateien. Die Standard-Bibliotheken befinden sich normalerweise im Verzeichnis *„MCPP:Lib"*, während in *„MCPP:Include"* sowohl die Standard-Includes von MaxonC++ als auch die jeweils aktuellsten Amiga-OS-Includes von Commodore liegen. Wenn Sie diese Dateinamen ändern wollen, sollten Sie wissen, was sie tun, und dieses Benutzerhandbuch aufmerksam gelesen haben.

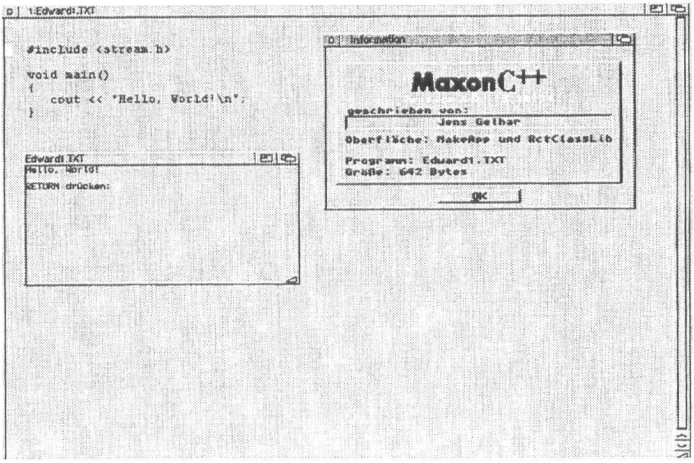
1.4 Erste Schritte mit MCPP

Die integrierte Entwicklungsumgebung *"MCPP"* kann wahlweise von der Workbench oder vom CLI gestartet werden. Dabei stellt *"MCPP"* noch keine vollständige Benutzeroberfläche dar, sondern lädt noch den Editor *"Edward"* nach. Folglich muß dieser Editor korrekt installiert sein. Wird der Edward nicht gefunden, steigt *"MCPP"* gleich mit einem entsprechenden Requester aus.

Nach dem Start landen Sie also automatisch im Editor. Hier können Sie Ihre C++-Programme schreiben und bearbeiten. Dem Edward ist ein separates Handbuch gewidmet, deshalb soll hier nicht näher auf die Editor-Funktionen eingegangen werden.Am besten machen Sie sich jetzt ein wenig mit dem Editor vertraut, indem Sie ein kleines Programm schreiben. Sie könnten z. B. folgendes unvermeidliche und traditionsreiche Prograämmchen abtippen:

```
#include <stream.h>
void main()
{ cout << "Hello, World!\n"; }
```

Nun haben Sie ein Programm als Quelltext vorliegen und können den Compiler ausprobieren. Zuerst sollten Sie den Text abspeichern, um ihm so einen sinnvollen Namen zu geben, mit dem der Compiler auch etwas anfangen kann. Dann wählen Sie einfach den Menüpunkt **„übersetzen"** aus der Menüleiste und warten ab, was geschieht.



Es öffnet sich ein Fenster, in dem *MCPP* einige mehr oder weniger kryptische Meldungen ausgibt. Tritt während der Übersetzung ein Fehler auf, wird er in diesem Fenster gemeldet und der entsprechende Quelltextbereich angezeigt. Sie können dann eine Fehlermeldung anwählen und mit den Gadgets ***„Anspringen"*** und ***„In Editor"*** direkt die entsprechende Textstelle im Editor anspringen. Wenn das System korrekt installiert wurde und ihr Programm ebenfalls fehlerlos ist, sollte das Fenster aber ebenso schnell verschwinden, wie es aufgetaucht ist. Nun können Sie die Menü-Funktion **„Ausführen"** anwählen und das Programm starten. Auch dabei öffnet sich ein Fenster, in dem (wer hätte das gedacht!) **"Hello, World!"** ausgegeben wird. Außerdem werden Sie freundlich aufgefordert, die <Return>-Taste zu drücken, denn sinnvollerweise schließt *MCPP* das Window nicht, bevor Sie Gelegenheit hatten, die Ausgabe zu lesen.

Sie können das Programm natürlich noch einmal starten. Falls Sie aber seit der letzten Übersetzung etwas am Quelltext verändert haben, wird automatisch der Compiler erneut gestartet.

2. Die integrierte Entwickungsumgebung

2.1 Programmstart und Optionen

Wenn Sie *MCPP* vom CLI bzw. aus einer Shell starten, können Sie optional einige Parameter angeben. Als erstes können Sie einen Dateinamen angeben, z. B.

```
run MCPP MCPP:Demos/hello.c
```

Sofern die angegebene Datei existiert und lesbar ist, wird sie in den Editor geladen. Da der Edward mehrere Texte verwalten kann, können Sie natürlich auch mehrere Dateinamen angeben.

Wenn Sie eine andere Konfigurationsdatei als *„s:MaxonC.config"* benutzen wollen, können sie ein zusätzliches Argument mit vorangestelltem `"-c"` angeben. Ein Beispiel:

```
run MCPP test.c -c MCPP:spezial.config
```

Im Gegensatz zur ersten Version von *MCPP* darf zwischen `"-c"` und dem Dateinamen jetzt auch ein Leerzeichen stehen. Da man sich spezielle Einstellungen jetzt aber auch in der Datei *"mcpp.SESSION"* abspeicher kann, gibt es nur noch wenig Gründe, dieses Feature zu benutzen.

MaxonC^{++} besitzt eine integrierte Projektverwaltung, allgemein bekannt als *„Make"*. Sie können gleich beim Programmstart eine solche Make-Datei öffnen, und zwar mit der Option `"-m"`:

```
run MCPP -m plot.makefile.
```

Natürlich lassen sich diese Optionen und Argumente in beliebiger Reihenfolge kombinieren, z. B.

```
MCPP fclass.c -m wagga.makefile -c df1:s/MaxonC.config.
```

Seit Version 3 besitzt *MCPP* einen eigenen ARexx-Port. Wenn Sie als einziges Argument `"-r"` angeben, also z. B.

```
run MCPP -r
```

starten MaxonC^{++} im ARexx-Modus. Dabei wird Edward nicht geladen. Statt dessen wird ein ARexx-Port namens `"MCPP"` eingerichtet, mit dem das System über ein ARexx-Script gesteuert werden kann. Auch für das Nachladen des Editors und die Kommunikation zwischen den beiden Programmen müssen Sie dann (über geeignete ARexx-Programme) selbst sorgen. So können Sie einen beliebigen - oder auch überhaupt keinen - Editor in die Umgebung einbinden.

Hinter der Option `"-r"` kann optional noch ein Portname angegeben werden, z. B,

```
run MCPP -r Wagga
```

In diesem Fall wird der ARexx-Port nicht `"MCPP"`, sondern `"WAGGA"` genannt. Man beachte, daß der Portname in Großbuchstaben umgewandelt wird. In jedem Fall aber gilt, daß MaxonC^{++} für eindeutige Portnamen sorgt. Falls z. B. schon ein Port `"MCPP"` existiert, werden so lange `"MCPP1"`,

`"MCPP2"`, `"MCPP3"` usw. ausprobiert, bis ein Name gefunden wird, den es im ganzen System noch nicht gibt.

Alles weitere über den ARexx-Port und was Sie damit anfangen können entnehmen Sie bitte Abschnitt 2.6.

2.2 Compiler und Editor - eine harmonische Beziehungskiste

Wie oben schon angedeutet, ist der Editor *Edward* ein eigenständiges Programm, das auch ohne den Compiler benutzt werden kann. *MCPP* lädt ihn einfach nach und startet ihn als zweiten Prozess, läßt aber keinen Zweifel daran aufkommen, daß er der Boß ist.

Weniger blumig ausgedrückt, richtet *MCPP* einen Port ein, über den sich Editor und Compiler Nachrichten zuschicken. Wird im Edward vom Benutzer eine Compiler-Funktion aufgerufen, etwa durch eines der Pull-Down-Menüs oder eine entsprechend belegte Tastenkombination, läßt Edward dem Compiler eine entsprechende Nachricht zukommen.

Ich erzähle ihnen diese ganzen implementatorischen Details, damit Sie verstehen können, wie Sie sich das MaxonC++-System selbst konfigurieren können. Der Schlüssel dazu ist die „*.DEF*"-Datei des Edward. Hier werden alle Menüs und Tastenbelegungen des Editors abgelegt, und hier muß irgendwie die Benutzerschnittstelle des Compilers integriert werden, auch wenn der Editor nicht die geringste Ahnung hat, was ein Compiler ist und wozu er gut ist.

Deshalb hat der Edward eine Spezial-Funktion namens `"SendAppComm"`, die als Argument wiederum eine Compiler-Funktion erwartet. Beispielsweise gibt es eine *MCPP*-Anweisung namens `"compile"`. Wird nun vom Benutzer, also Ihnen, im Edward ein Menü ausgewählt, das mit der Funktion `SendAppComm "compile"` belegt ist, schickt Edward das `"compile"` als Nachricht an den *MCPP* und kümmert sich dann nicht weiter drum. *MCPP* dagegen erhält diese Anweisung, fordert vom Edward noch ein paar zusätzliche Daten an und beginnt dann freudig mit der Übersetzung. Natürlich kann man nicht nur ein Menü, sondern auch eine Taste oder Tastenkombination mit diesen `"SendAppComm"-Anweisungen` belegen.

Auf diese Weise sind der Compiler und der Editor vollkommen voneinander abgekoppelt, und *Edward* braucht nichts über *MCPP* zu wissen. Nun kommt es natürlich vor, daß Sie den Edward einmal „einfach so" ohne Compiler benutzen wollen. Deshalb läßt Edward einen kleinen „Preprozessor" auf die „*.DEF*"-Datei los und filtert je nach Modus Teile aus dem Text aus - ähnlich wie ein C-Preprozessor bei `"#ifdef"`-Zeilen. Auch dieses Feature ist in ihrem Edward-Handbuch beschrieben. *MCPP* teilt dem *Edward* mit, daß ein Symbol namens `"MCPP3"` definiert sein soll. Nun können Sie Dinge wie `"#ifdef MCPP3"` in Ihr *DEF-File* schreiben, um Definitionen je nach Modus ein- oder auszublenden. Insbesondere werden Sie sich um sämtliche Menü-Definitionen des Compilers eine `"#ifdef MCPP3 ... #endif"`-Klammer legen wollen, wie es auch in der mitgelieferten Standard-Datei der Fall ist.

Langer Rede schwacher Sinn: Wenn im folgenden von irgendwelchen Menüs die Rede ist, werden dabei jedesmal Hinweise wie „...oder wie auch immer Sie diese Funktion konfiguriert haben" unter-

schlage. Gemeint ist damit jeweils immer die Funktion, die in der Standard-Konfiguration auf dem entsprechenden Menü liegt.

Es gibt übrigens standardmäßig folgende Menüs, mit den nachstehend angegebenen Compiler-Funktionsnamen:

Compiler

- Übersetzen	compile
- Compile Datei	compfile
- Ausführen	run
- Abbrechen	break
- Debugger	debug
- Exe-Datei	exefile
- Objektdatei	objectfile
- Information	info
- Fehlermeldungen	errors

Make

- Projektverwaltung	make
- Neues Makefile	newmk
- Makefile laden	loadmk
- Text aus Editor aufnehmen	addedittx
- Text und Objekt aufnehmen	addtxobj
- Make verlassen	leavemk

Optionen

- System	opsys
- Compiler	opcomp
- Linker	oplink
- Optionen sichern	saveconf

2.3 Das Konzept

Auch auf die Gefahr hin, daß ich Sie damit langweilen könnte, möchte ich trotzdem noch ein paar grundsätzliche Anmerkungen loswerden. Prinzipiell kann MCPP in zwei verschiedenen Modi arbeiten: Im **„normalen"**-Modus, in dem man vor allen kleinere Programme (mit nur einem Modul) schreibt, und im **„Make"**-Modus. Da der Projekverwaltung „Make" ein eigenes Kapitel gewidnet ist, befassen wir uns zunächst nur mit dem Standard-Modus, der natürlich auch aktiv ist, wenn Sie das MaxonC^{++}-System starten.

Auch wenn *Edward* beliebig viele Texte gleichzeitig bearbeiten kann, verwaltet *MCPP* immer nur ein Programm. Das hat zum einen Gründe, die mit dem Speicherplatz zusammenhängen, und resultiert

zum anderen aus der Beobachtung, daß das in der Praxis nicht besonders relevant ist, weil man normalerweise zwar durchaus mit mehreren Texten, aber an nur einem Programm arbeitet.

Die Funktion `"compile"` übersetzt stets den Text im gerade aktiven Fenster. Bei `"run"` wird immer zuerst folgendes geprüft:

- **Liegt überhaupt ein übersetztes Programm vor?**
- **Befindet sich der Quelltext dieses Programms im aktiven Editor-Fenster?**
- **Blieb er seit der letzten Übersetzung unverändert?**

Falls eine der drei Fragen mit „Nein" beantwortet werden muß, wird die `"compile"-Funktion` aufgerufen und damit der Text im aktiven Fenster übersetzt.

So, das sollte gesagt werden, damit Sie beim Rumprobieren mit MaxonC++ (was Sie wahrscheinlich schon längst getan haben) nicht in vermeidbare Verwirrung stürzen (was dabei hoffentlich nicht geschehen ist). Und nun folgt endlich eine ausführliche Beschreibung der Funktionen von MaxonC++:

2.4 Compiler-Funktionen

2.4.1 Programme übersetzen

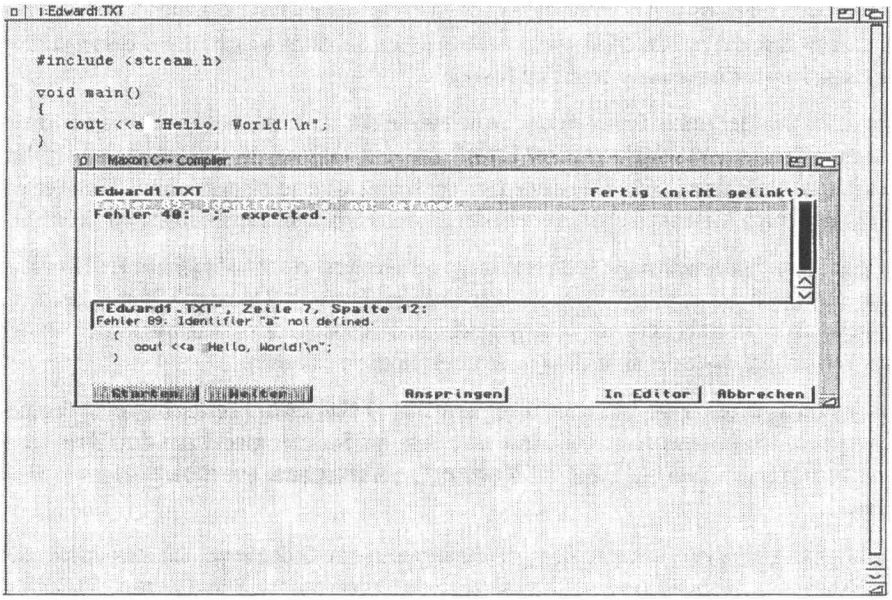
2.4.1.1 Das Compiler-Fenster

Am besten fangen wir mit der Funktion an, wegen der Sie MaxonC++ überhaupt gekauft haben, nämlich dem Übersetzen von Programmen.

Wenn Sie diese Funktion auswählen, wird - je nach Modus - entweder das Programm im gerade aktiven Editor-Fenster übersetzt, oder das integrierte „Make" entscheidet selbst, welche Objektdateien neu übersetzt werden sollen. In jedem Fall öffnet sich aber ein Fenster, das in fünf Bereiche aufgeteilt ist.

Am oberen Rand befindet sich die Statuszeile. Hier wird stets links der Name der gerade übersetzten Datei angezeigt, während *MCPP* rechts sagt, was gerade getan wird bzw. zuletzt getan wurde.

Am besten probieren Sie es einmal aus: Laden Sie einen Quelltext (z. B. eines der Beispielprogramme) in den Editor und wählen Sie den Menüpunkt **"Übersetzen"** aus. Wenn alles klappt, wird oben rechts in schneller Folge (bei kurzen Programmen wohl zu schnell zum Mitlesen) das folgende ausgegeben:



- **"Übersetze":**
  Der Compiler liest und analysiert das Programm.

- **"Keine Fehler gefunden" bzw. "X Fehler gefunden":**
  Nachdem der Quelltext gelesen und auf seine Korrektheit geprüft wurde, erzeugt der Compiler nun den Code dazu.

- **"Programm wird gelinkt":**
  Der Linker bindet das erzeugte Programm mit den nötigen Bibliotheksfunktionen zusammen.

- **"Fertig":**
  Das Programm ist übersetzt und ausführbar. Wurden keine Fehler oder Warnungen ausgegeben, verschwindet das Window bei dieser Meldung automatisch.

Natürlich kann dabei beliebig viel schief gehen, beispielsweise könnte das Programm fehlerhaft sein. Fehlermeldungen und Warnungen aller Art werden in dem Rahmen unterhalb der Statuszeile ausgegeben. *MCPP* gibt immer höchstens so viele Fehler aus, wie in diesen Rahmen passen (das sind zehn Stück) und wartet auf eine Reaktion des Benutzers, falls die Höchstzahl erreicht wurde.

Eine solche einzeilige Fehlermeldung allein ist meist natürlich nicht aussagekräftig genug. Deshalb kann sich der Programmierer jeweils zu einer Meldung ausführlichere Informationen anzeigen lassen.

Unterhalb des Fehlermeldungs-Rahmen befindet sich ein weiterer Rahmen, in den normalerweise zwei Zeilen ausgegeben werden. Darin stehen Quelltextdatei, Zeilen- und Spaltennummer des gera-

de angewählten Fehlers und noch einmal eine Wiederholung der schon im oberen Rahmen ausgegebenen Fehlermeldung. Unterhalb dieser Anzeige sehen Sie einen Ausschnitt aus dem Quelltext mit einem Pseudo-Cursor an der fraglichen Position.

Wenn allerdings der Linker Fehler meldet, steht anstelle der Sourceposition (was weiß ein Linker schon von Quelltexten!) schlicht **„Linker Error"**, und dort, wo sonst der Quelltext-Ausschnitt dargestellt wird, steht eine eventuell verkürzte Liste der beim Zusammenbinden aufgetretenen Fehler, also hauptsächlich Namen von undefinierten oder gleich mehrfach definierten Linker-Symbolen.

Die eigentliche Fehlermeldungen sind recht knapp gehalten und in der vorliegenden Version obendrein auch noch auf Englisch. Wenn Sie aber eine der Tasten <H> oder <Help> drücken, erscheint ein HotHelp-Fenster mit weitergehenden und deutschsprachigen Informationen. In der Light-Version geht das leider nicht, denn da ist ja kein HotHelp enthalten.

Weniger häufig treten beim Übersetzen unter dem Titel **„Fatal Error"** Fehler bei der Codegenerierung oder beim Assemblieren auf. Meist sind diese auf Speichermangel zurückzuführen, und dementsprechend heißen sie **„Out of Memory"**, **„Workspace overflow"** oder wie auch immer.

Last not least gibt es ganz unten im Compiler-Fenster noch eine Gadgetleiste. Zunächst ist hier nur das Gadget ganz rechts mit der Aufschrift ***„Abbrechen"*** anwählbar. Bei Bedarf werden die übrigen Gadgets aktiviert.

2.4.1.2 Abbruch und Ausstieg

Im Gegensatz zu einfachen Kommandozeilen-Compilern besitzt das MaxonC++-Fenster eine Vielzahl von Knöpfen und Funktionen, an denen der C++-Programmierer herumspielen kann. Alle Gadget-Funktionen sind, um den Bedürfnissen der „echten Programmierer" entgegenzukommen, auch durch Tasten oder Tastenkombinationen bedienbar.

Beim Start des Compilers können Sie im Compiler-Fenster zunächst nur zwei Dinge tun: Das Fenster schließen oder die Übersetzung abbrechen.

Mit einem Klick auf das ***Schließ***-Gadget wird die Übersetzung abgebrochen und das Fenster geschlossen. Die Taste <ESC> hat die gleiche Wirkung.

Außerdem gibt es unten rechts das bereits erwähnte ***„Abbrechen"***-Gadget. Man benutzt es, wenn man den Compiler zwar abbrechen, aber noch irgendwelche im Fenster ausgegebenen Meldungen in Ruhe lesen möchte. Die kanonische Tastenkombination <Control>+<C> entspricht diesem Gadget.

Falls der Compiler aber schon fertig ist, wird das Compiler-Fenster bei einem Klick auf ***„Abbrechen"*** geschlossen.

2.4.1.3 Auswahl von Fehlermeldungen

Laut Murphy geht bei so ziemlich jedem Compiler-Lauf etwas schief, und meist sogar etwas mehr als nur „etwas". Deshalb bietet das Compiler-Fenster Platz für bis zu zehn Fehlermeldungen auf einmal.

Eine dieser jeweils einzeiligen Meldungen ist immer invertiert und somit hervorgehoben, und zu genau dieser Meldung werden unterhalb des Rahmens zusätzliche Informationen angezeigt. Mit einem Mausklick auf eine Fehlerzeile macht man diese zur gerade privilegierten, invertierten Meldung.

Das geht aber auch über die Tastatur: Mit der <Leertaste> durchläuft man die Fehlermeldungen von oben nach unten, und mit <Backspace> in die andere Richtung. Alternativ können Sie auch die Cursortasten <Auf> und <Ab> benutzen.

2.4.1.4 Vom Compiler in den Editor

Der besondere Witz an einer integrierten Umgebung ist ja das perfekte Zusammenspiel von Compiler und Editor, insbesondere das automatische Anspringen von Fehlerstellen.

Das funktioniert bei MaxonC++ denkbar einfach: Sobald eine Fehlermeldung im Compiler-Fenster ausgegeben wird, werden auch zwei Gadgets mit der Aufschrift ***„Anspringen"*** bzw. ***„In Editor"*** aktiviert. Nun wählen Sie sich wie oben beschrieben Ihre Wunsch-Fehlermeldung aus (das geht übrigens auch, während der Compiler noch arbeitet) und klicken ***„Anspringen"*** an. Prompt lädt Edward die zugehörige Quelltextdatei (falls der Text nicht ohnehin schon im Editor vorliegt) und setzt den Cursor an die bemängelte Position. Nun können Sie den Fehler berichtigen - vorausgesetzt natürlich, der Compiler läuft nicht noch und hat ein „Lock" auf diesen Text eingerichtet. Das Compiler-Fenster wird dabei nicht geschlossen, so daß Sie noch weitere Fehlerstellen anspringen können.

Die Integration von Compiler und Editor geht allerdings nicht so weit, daß im Compiler-Window Quelltext-Veränderungen beachtet würden. Wenn Sie z. B. zuerst einen Fehler berichtigen und dabei einige Zeilen einfügen oder löschen müssen, merkt *MCPP* nichts davon. Sollten Sie also anschließend ein weiter unten im Quelltext gelegene Fehlerstelle anspringen lassen, landen Sie gnadenlos in der Zeile mit der Nummer, in der der Fehler lag, als er bemerkt wurde - und durch die inzwischen vorgenommenen Änderungen stimmt diese Zeilennummer inzwischen vielleicht nicht mehr, und Sie landen einige Zeilen zu hoch oder tief. Das dürfte aber in der Praxis kaum ein Problem darstellen, denn im Compiler-Fenster wird immer noch der ursprüngliche Quelltext mit dem Cursor an der richtigen Stelle angezeigt, und durch einen kurzen Textvergleich zwischen Compiler- und Editorfenster können Sie schnell die wirkliche Fehlerposition „zu Fuß" ansteuern. Außerdem ändert man in der Praxis bei Compilerfehlern meist sowieso nur die fragliche Zeile, ohne großartiges Löschen und Einfügen von Zeilen.

Das Gadget ***„Anspringen"*** ist nebenbei bemerkt mit dem Shortcut <A> belegt.

Manchmal tritt während der Übersetzung nur ein einziger Fehler auf, der vielleicht noch den einen oder anderen Folge-Fehler nach sich zieht, aber trotzdem nur eine einzige Änderung am Quelltext notwendig macht. Deshalb gibt es auch noch das Gadget ***„In Editor"***. Hier wird im Editor die gerade angewählte Fehlerposition angesprungen (genau wie beim ***„Anspringen"***-Gadget) und gleichzeitig das Compiler-Fenster geschlossen - wobei natürlich auch der Compiler abgebrochen wird, sollte er noch am arbeiten sein.

Alternativ können Sie auch die Tasten <Return> oder <E> drücken, und ebenso gut können sie auch mit der Maus einen ***Doppelklick*** auf die Fehlermeldung machen.

2.4.1.5 „Weiter"

Treten beim Übersetzen bzw. Linken mehr Fehler auf, als im Rahmen dargestellt werden können (das sind bekanntlich zehn Stück), wartet der Compiler erst einmal ab, und das ***„Weiter"***-Gadget wird anwählbar. Nach einem Klick auf selbiges läuft der Compiler wieder weiter, bis erneut zehn Fehlermeldungen voll sind. Meist ist es aber nicht unbedingt sinnvoll, nach mehr als zehn Fehlern noch weiter zu übersetzen, denn normalerweise kommt dabei nicht mehr so ganz viel sinnvolles 'raus. Deshalb wird man in der Regel das ***„Weiter"***-Gadget meiden und lieber die Fehler korrigieren.

2.4.1.6 Programmstart aus dem Compiler-Fenster

Wenn MaxonC++ bei der Übersetzung Fehler oder Warnungen ausgegeben hat, aber trotzdem ein lauffähiges Programm erzeugen und linken konnte, taucht unten rechts das ***„Starten"***-Gadget auf. Mit einem beherzten Klick darauf können Sie das Programm starten - aber vorher sollten Sie sich genau überlegen, ob die gemeldeten Fehler beim Starten des Programms nicht zu einem Guru oder ähnlich unangenehmen Effekten führen können! Bedenken Sie z. B., daß für eine Programmzeile, die einen groben Fehler enthält, in der Regel keinen Code erzeugt wird - und in einer „Ohne Netz und doppelten Boden"-Programmiersprache wie C++ bzw. C kann auch eine fehlende Anweisung fatale Folgen haben...

2.4.1.7 Erneutes öffnen des Compiler-Fenster

Wenn Sie das Compiler-Fenster geschlossen haben und es doch noch einmal sehen wollen, können Sie es mit dem Menüounkt **"Fehlermeldungen"** noch einmal öffnen. Wie der Name schon andeutet, sind dann alle Fehlermeldungen der letzten Compilierung wieder sichbar, und auch sonst präsentiert sich das Fenster in dem Zustand, in dem es sich beim Schließen befand. Ein Tip für Maus-Hasser: Auch wenn das Compiler-Fenster noch geöffnet ist und der Compiler vielleicht sogar noch arbeitet, können Sie es mit dem Menü - oder besser mit einer sinnvoll gewählten Tastenbelegung - jederzeit aktivieren und in den Vordergrund holen. So können Sie zwischen Editor- und Compiler-Fenster hin- und herwechseln, ohne zur Maus greifen zu müssen.

2.4.2 Ausführen und Abbrechen

Die zweite wichtige Funktion aus dem **„Compiler"-Menü** heißt **„Ausführen"** und tut auch genau das: Ein C++-Programm wird gestartet.

Vorher prüft *MCPP* allerdings noch, ob nicht vielleicht erst noch der Compiler konsultiert werden muß, und startet selbigen automatisch.

Anschließend wird ein Text-Fenster für die Ein- und Ausgaben des Programms geöffnet. Falls das Programm tatsächlich etwas ausgegeben hat, erscheint nach erfolgreicher Programmausführung die Aufforderung **„Press Return"** - Sie sollen also die <Return>-Taste drücken und so bestätigen, daß Sie die Ausgaben des Programms auch gelesen haben.

Während das Programm läuft, sind verschiedene Funktionen des Systems gesperrt, z. B. darf man es (trotz Multitasking) nicht ein zweites mal gleichzeitig starten, oder man kann den Compiler nicht laufen lassen, weil man dem laufenden Programm ja dadurch den Code unter den Füßen wegziehen bzw. überschreiben würde. Das ist sicher sinnvoll und stört wohl nicht weiter - es sei denn, das Programm ist irgendwo hängengeblieben und läßt sich nicht normal beenden.

Gegen „normale" Endlosschleifen hilft die Compileroption ***„Unterbrechen"***. Schwieriger wird es dagegen, wenn der Prozeß abgestürzt ist, auf ein Ereignis wartet, das nie eintreten kann oder die erwähnte Option ***„Unterbrechen"*** einfach nicht benutzt wurde. Deshalb (und nur für solche Fälle!) gibt es im **Compiler-Menü** die Funktion **„Abbrechen"**, die den gestarteten Prozeß schlicht abtötet und dann das Ausgabe-Fenster schließt. Auf den ersten Blick eine praktische Lösung, aber leider gibt es auf dem Amiga (im Gegensatz z. B. zu Unix) keine saubere Möglichkeit, einen Prozeß aus dem System zu entfernen. Insbesondere werden dabei keine Ressourcen (reservierter Speicher, geöffnete Dateien, Screens und Fenster etc.) freigegeben. Sie müssen davon ausgehen, daß sie auf diese Weise freien Speicher verlieren, wenn Sie einen Prozeß auf diese brutale Art abbrechen. Aber das ist im Zweifel immer noch angenehmer, als das MCPP-System neu zu starten oder gar den Rechner zu booten.

2.4.3 Aufruf des Debuggers

Im Compiler-Menü finden Sie auch die Funktion **„Debugger"**. Wenn Sie Besitzer der Developer-Version sind, können Sie hier den Debugger *MaxonSDB* aufrufen.Um den Speicherverbrauch des MaxonC^{++} Systems nicht unnötig zu vergrößern, wurde der Debugger nicht wirklich in die Entwicklungsumgebung integriert, sondern muß bei Bedarf nachgeladen werden. Im „System"-Einstellfenster können Sie die dafür notwendigen Einstellungen vornehmen, falls Sie mit den Defaultwerten nicht zufrieden sind.

Bei Aufruf des Menüpunkts **„Debugger"** prüft *MCPP* zunächst, ob bereits eine ausführbare Programmdatei geschrieben wurde - falls nicht, geschieht dies jetzt automatisch. Dabei wird auch auf jeden Fall eine *„.mdbg"-Datei* erzeugt, aus der der Debugger erfährt, aus welchen Objektdateien das Programm besteht. Die ebenfalls wichtigen *„.mdbi"-Dateien* können jetzt aber nicht mehr erzeugt werden, da sie nur während der Übersetzung geschrieben werden können. *MaxonSDB* wird Ihnen das Fehlen solcher Dateien ggf. melden. Mein Tip: Wenn Sie den Debugger besitzen und prinzipiell auch benutzen möchten, sollten sie die Option für die Erzeugung der Debugger-Dateien generell eingeschaltet lassen.

2.4.4 Exe- und Objektdateien

Ungeachtet allen damit verbundenen Komforts werden Sie ihre Programme wohl nicht immer nur aus der integrierten Umgebung starten wollen. Deshalb gibt es im **„Compiler"-Menü** zwei Funktionen, die das übersetzte Programm in Dateien schreiben:

☞ **„Exe-Datei"** erzeugt eine ausführbare Programmdatei - vorausgesetzt natürlich, ein Programm ist korrekt übersetzt und gelinkt worden.

☞ **„Objekt-Datei"** schreibt zur zuletzt compilierten Übersetzungseinheit eine Objektdatei, aus der man später z. B. mit dem `"mcppc"` ein ausführbares Programm erzeugen kann. Im „Make"-Modus ist diese Funktion nicht unbedingt sinnvoll, denn dort werden diese Objektdateien ja beim Compilieren automatisch erzeugt.

2.4.5 Daten und Informationen

Mit dem Menüpunkt **„Information"** öffnet man ein Fenster, in dem Informationen über den Status von *MCPP* angezeigt und ständig aktualisiert werden. Neben dem üblichen Rhabarber über Autorschaft und Copyright (klicken Sie doch einfach mal auf den Namen des Autors...) können Sie hier bei Bedarf nachlesen, welchen Quelltext Sie zuletzt übersetzt oder welche Make-Datei gerade geladen ist. Liegt ein übersetztes Programm im Speicher vor, wird auch seine Größe angegeben. Dabei handelt es sich genaugenommen um den Platzbedarf des Programms, wenn es in den Speicher geladen wird, und nicht um die Größe der Exe-Datei.

2.5 Features und Optionen

2.5.1 Ein kurzer Überblick

Für viele Programmierer ist das Schönste an einem Compiler, daß man so viele Optionen und Switches hat, an denen man herumspielen kann. Bei *MCPP* nimmt man sämtliche dieser Einstellungen über Dialoge vor, deren es genau drei gibt: Einen – **„System"** – für allgemeine Einstellungen, die in irgendeiner Weise mit dem MCPP-System zu tun haben, einen weiteren für die unvermeidlichen **Compiler-Optionen** und schließlich einen eigenen Dialog für die **Linker-Steuerung**.

Wie Sie sich vielleicht denken können, beherrschen *MCPP* und die Kommandozeilen-Version *„mcppc"* praktisch die selben Einstellungen und Optionen, nur die Bedienung ist eben unterschiedlich. Deshalb sollten Sie sich dieses Kapitel auch dann durchlesen, wenn Sie ausschließlich den *„mcppc"* benutzen wollen, denn bei der Beschreibung der Optionen im *„mcppc"*-Kapitel wird hauptsächlich das „Wie" der Einstellungen beschrieben, während ausschließlich hier ausführlich erläutert wird, was diese Optionen genau bedeuten.

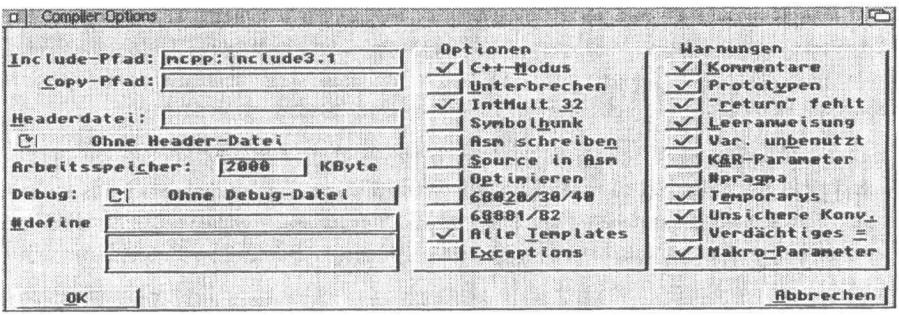
Zurück zum *MCPP*: Wenn Sie wollen, können Sie alle drei Dialoge gleichzeitig öffnen. In jedem der Fenster finden Sie am unteren Rand die obligatorischen Gadgets ***„OK"*** und ***„Abbruch"***, mit denen Sie die vorgenommenen Einstellungen bestätigen bzw. Sie wieder verwerfen können.

2.5.2 Compiler-Optionen

2.5.2.1 Include-Dateien

Wie eigentlich alle modernen Programmiersprachen, sind C und C++ ziemlich „dumm": es gibt für den Compiler keinerlei vordefinierte Funktionen (während z. B. in Pascal `"WriteLn"` oder in BASIC `"print"` stets zur Verfügung stehen). Vielmehr stehen Definitionen für die Standard-Funktionen in diversen Includedateien, die man explizit einbinden muß.

Vielleicht wissen Sie schon, daß es dafür zwei verschiedene Möglichkeiten gibt: Schließt man den Dateinamen `"so"` in Anführungszeichen ein, sucht der Compiler (oder der Preprozessor, um genauer zu sein) vom aktuellen Verzeichnis aus, während die `<so>` in spitze Klammern gesetzten Dateinamen Standard-Includedateien bezeichnen und in einem ganz bestimmten Verzeichnis gesucht werden. Von genau diesem Verzeichnis, und wie man es festlegt, soll hier die Rede sein.



Im **"Compiler"**-Dialogfenster legen Sie mit den beiden Textgadgets oben links genau dieses Verzeichnis fest. Wenn Sie nur das obere Gadget benutzen, dürfte die Funktionsweise klar sein: Der Compiler sucht die Include-Dateien im dort angegebenen Verzeichnis. Für alle, die sich um den RAM-Bedarf keine allzu großen Sorgen machen müssen, gibt es auch noch die Möglichkeit, Includes umkopieren und dabei leicht vorcompilieren zu lassen. Dazu dient das zweite Stringgadget: Hier können Sie ein zweites Verzeichnis angeben, das vorzugsweise in der *"RAM:"*-Disk liegen sollte, aber natürlich nicht muß. Der Compiler sucht dann zuerst in diesem zweiten Verzeichnis nach den Includedateien. Findet er sie dort nicht, greift er auf das andere Verzeichnis zu und kopiert die benutzten Dateien von dort automatisch um. Dabei werden sie aber auch noch leicht „vorcompiliert" (nämlich als Token abgespeichert), wodurch alle weiteren Übersetzungen etwas beschleunigt werden. Durch Angabe eines zweiten Pfades können Sie die Dateien also gleichzeitig auf ein schnelleres Speichermedium umkopieren und vorcompilieren lassen.

Übrigens legt MaxonC^{++} vor dem Umkopieren/Vorcompilieren bei Bedarf selbständig die benötigten Unterverzeichnisse an.

Vor einer kleinen Heimtücke des Systems sei noch gewarnt: Achten Sie bitte darauf, daß Sie weder an den Anfang noch an das Ende der Dateinamen versehentlich irgendwelche Leerzeichen anhängen! *MCPP* versucht sonst gänzlich ungerührt, auf die angegebenen Pfade einschließlich der überflüssigen Spaces zuzugreifen, und da wird er naturgemäß nicht allzu viel finden. Außerdem sollten Sie sich, wenn Sie in die RAM-Disk vorcompilieren lassen, nach dem Verlassen des MCPP-Systems auch nicht über einen scheinbaren Speicherverlust wundern - das liegt einzig und allein an den vorcompilierten Daten in der RAM-Disk, die sich bei Bedarf leicht mit einem CLI-Befehl wie

```
delete ram:include all quiet
```

löschen lassen.

2.5.2.2 Vorcompilierte Header-Dateien

MaxonC++ kann Teile von Programmen vorcompilieren, d. h. an einer vom Benutzer frei wählbaren Stelle kann der Compiler alle seine internen Daten - und damit die bisher gelesenen Deklarationen und Definitionen - in einer Datei abspeichern. Beim nächsten Compilerdurchlauf kann diese Datei sehr schnell geladen und so viel Zeit gespart werden.

Dieses Feature ist dafür gedacht, die Programmentwicklung zu beschleunigen, wenn sehr viele Header-Dateien benutzt werden, z. B. die AmigaOS-Includes oder die MaxonC++ Klassenbibliothek "*EasyObjects*". Damit der Compiler aber weiß, wo die Header aufhören und Ihr eigenes Programm anfängt, muß man ihm diese Stelle mit einer Zeile `"#pragma header"` markieren.

Am besten schreiben Sie in den Quelltext jeder Überstzungseinheit zunächst die `"#include's"` all jender Headerdateien, die Sie nicht selbst geschrieben haben oder an denen sich aus anderen Gründen nichts mehr ändern wird. Dann folgt die Zeile `"#pragma header"`, und anschließend folgen die Include-Anweisungen für Ihre eigenen Header-Dateien und der Rest Ihres Programms.

Normalerweise hat `"#pragma header"` keinerlei Wirkung. Wenn Sie aber im Textfeld "Header-Datei" des Compiler-Dialogfensters einen beliebigen Dateinamen eintragen und das Gadget darunter auf "Header-Datei schreiben" stellen, hält der Compiler, sobald das Pragma erreicht wird, kurz inne und schreibt alle Definitonen und Deklarationen in die namgegebene Datei. Danach arbeitet er weiter, als sei nichts geschehen.

Vor dem nächsten Compilerdurchlauf schalten Sie dann einfach um auf "Header-Datei lesen", wobei Sie den Dateinamen unverändert belassen. Dann liest der Compiler die vorcompilierten Header aus der angegebenen Datei, sucht im Quelltext nach dem `"#pragma header"` und setzt dort mit seiner Übersetzungstätigkeit ein.

Da alles, was vor dem `"#pragma header"` steht, komplett ignoriert wird, muß das Pragma folglich in der Haupt-Quelltextdatei der Übersetzungseinheit - und nicht etwa in einer Includedatei - stehen.

Denken Sie bitte daran, daß *MCPPC* nicht prüft, ob der Inhalt der vorcompilierten Header-Dateien überhaupt noch mit dem ursprünglichen Quelltext übereinstimmt. Deshalb empfehle ich noch einmal dringend, nur solche Header-Dateien vorzucompilieren, an denen sich während der Programmentwicklung nichts mehr ändert.

Wundern Sie sich bitte auch nicht, wenn er Compiler vor dem eigentlichen Schreiben der Header-Dateien eine ganze Weile scheinbar überhaupt nichts tut. In dieser Zeit sammelt und sortiert er seine Daten, und das kann dauern. An Speicher zieht er sich dabei auch so einiges rein, so daß Sie bei knappem RAM-Ausbau und/oder großen Header-Files eventuell die integrierte Umgebung verlassen und auf den Shell-Compiler "*mcppc*" ausweichen müssen. Beim späteren Laden der so erzeugten Header-Dateien werden Sie aber keine derartigen Probleme haben, denn gegenüber der "normalen" Übersetzung spart man mit vorcompilierten Headern nicht nur eine memge Zeit, sondern auch noch ein bißchen Speicherplatz.

2.5.2.3 Der Arbeitsspeicher des Compilers

Hin und wieder braucht der MaxonC++ Compiler einen zusammenhängenden Speicherbereich, nämlich bei der Codegenerierung und Assemblierung. Außerdem legt er nach erfolgreichem Linken in diesem Speicherblock eine lauffähige Kopie des erzeugten Programms ab.

Was Sie damit zu schaffen haben?

Eigentlich nichts, aber leider weiß MaxonC++ nicht im Voraus, wieviel Speicher er dafür brauchen wird. Der Compiler kann später höchstens einen Überlauf feststellen, aber vergrößern kann er diesen ***„Workspace"*** genannten Speicherblock dann nicht mehr, weshalb er mit einer Meldung wie **„Workspace Overflow"** aussteigt.

Deshalb müssen Sie leider persönlich festlegen, wie groß dieser Workspace sein soll, und zwar mit den entsprechendem Integer-Gadget im Compiler-Requester. Der Defaultwert von 80 kByte dürfte übrigens für die allermeisten Zwecke ausreichen, und wenn Sie mit RAM etwas knapp sind, können Sie ruhig versuchen, die Workspace-Größe weiter zu verringern. Sollte der Compiler dagegen einmal **„Workspace Overflow"** melden, setzen Sie die Größe einfach etwas herauf und versuchen es noch einmal.

2.5.2.4 Debug-Datei

Das Gadget ***"Debug"*** ist für Sie nur von Interesse, wenn Sie den MaxonSDB besitzen. Der Compiler kann zu jeder Übersetzungseinheit eine *"MDBI"*-Datei anlegen, aus welcher der Debugger umfangreiche Informationen entnehmen kann.

Diese können aber "mehr unfangreich" oder "weniger umfangreich" sein: Wenn Sie sich für die Einstellung ***"Komplette Debug-Datei"*** entscheiden, taucht praktisch jede Deklaration des Programms in der Datei auf und steht dann auch im Debugger zur Verfügung. Bei einem Programm, das z. B. die unvermeidlichen AmigaOS-Includes enthält, tauchen dann die ganzen Betriebssystem-Strukturen in der *MDBI*-Datei auf - und das bläht diese Dateien mächtig auf.

Da man sich beim Debugging normalerweise mehr für sein eigenes Programm als für Interna des Betriebssystems oder irgendwelcher Klassebibliotheken interessiert, sollte man sich besser für die "kleine" Variante der *MDBI*-Dateien entscheiden. Dann werden Deklarationen aus Standard-Includedateien (das sind die, die man mit den `<spitzen Klammern>` includiert) nicht an den Debugger weitergereicht. Das spart Zeit und Speicherplatz, und zwar gleichermaßen beim Compilieren des Programms wie auch bei der Debugger-Sitzung.

2.5.2.5 Vordefinieren von Symbolen

Mit den drei Gadets, die mit ***„#define"*** überschrieben sind, können Sie bis zu drei Preprozessor-Symbole vordefinieren. Jeder Eintrag in einem dieser Textgadgets wirkt dann so, als stände er mit einem vorangestellten `"#define"` am Anfang jedes übersetzten Quelltexts. Dieses Feature entspricht also der `"-d"`-Option von „mcppc".

Ein Beispiel: Sie haben als vorsichtiger Programmierer die Includedatei `"<assert.h>"` eingebunden und diverse Sicherheitsabfragen in Ihr Programm gesetzt - übrigens ein löbliches Unterfangen.

Um diese Abfragen automatisch aus dem Programm auszublenden, müssen Sie dann `"#define NDEBUG"` irgendwo an den Anfang des Quelltexts setzen - oder ganz einfach `"NDEBUG"` in eines der drei *„#define"*-Gadgets schreiben.

Sie können Symbolen aber nicht nur „einfach so" definieren, sondern ihnen auch Werte zuweisen. Ein Gadget-Eintrag wie `"N 100"` definiert das Symbol `"N"` als `"100"`. Leider akzeptiert der Compiler bei solchen Vorab-Definitionen immer nur ein Token pro Symbol: Der Eintrag `"ANZAHL (17+4)"` belegt `"ANZAHL"` mit dem Zeichen `"("` und ignoriert den Rest einfach. Aber komplexe Definitionen wird man normalerweise ohnehin in den Quelltext schreiben, während derartige „externe" Symboldefinitionen eher für irgendwelche Flags wie `"NDEBUG"` benutzt werden.

2.5.2.6 Diverse kleine, aber trotzdem wichtige Schalter

Nahezu unübersehbar befinden sich im Compiler-Dialog zwei Spalten mit diversen Switches - also so etwas wie ein virtuelles Mäuseklavier. Die linke Spalte ist mit *„Optionen"* überschrieben und enthält demzufolge eine mehr oder weniger geordnete Ansammlung von ein- und ausschaltbaren Compileroptionen.

2.5.2.6.1 C++

Mit dem Gadget *„C++"* können Sie wählen, ob MaxonC++ als echter C++-Compiler oder kastriert im ANSI C-Modus arbeiten soll. Ich hoffe doch stark, daß Sie die C++-Option nur in Ausnahmefällen ausschalten werden. Sie können diese Umschaltung übrigens auch mit einer `"#pragma"`-Zeile direkt im Quelltext vornehmen. Näheres dazu erfahren Sie im Kapitel über den Preprozessor (5.5.1 im Tutorial).

2.5.2.6.2 Unterbrechen

Mit *„Unterbrechen"* legen Sie fest, ob der Compiler im erzeugten Code Breakpoints setzen soll. Wenn diese Option angewählt ist, baut er in jede Schleife eine Abfrage der Tastenkombination <Ctrl>+<C> ein.

2.5.2.6.3 IntMult 32

Der 68000er-Prozessor kann leider keine Langworte multiplizieren und dividieren, sondern nur zwei 16-Bit-Werte zu einem 32-Bit Ergebnis multiplizieren und eine 32-Bit-Zahl durch ein 16-Bit-Wort dividieren, wobei Quotient und Rest jeweils 16 Bit breit sind. Für numerische Anwendungen ist das natürlich nicht so schön, weshalb MaxonC++ optional solche Operationen durch Bibliotheksfunktionen ersetzen kann.

Diese korrekt rechnenden Funktionen werden benutzt, wenn das Gadget *„IntMult 32"* aktiviert ist, und kosten natürlich einiges an Rechenzeit. Dieser Laufzeit-Aufwand ist aber in der Praxis nicht so gravierend, und für korrekte Ergebnisse ist uns schließlich nichts zu teuer.

Falls Sie aber die 68020-Option benutzen, ist der Schalter *„IntMult 32"* bedeutungslos.

2.5.2.6.4 Debug-Datei

Die Compileroption ***„Debug-Datei"*** ist für Sie nur von Interesse, wenn Sie den Maxon Debugger besitzen. Der Compiler legt dann zu jeder Übersetzungseinheit eine Datei an, aus welcher der Debugger umfangreiche Informationen entnehmen kann.

2.5.2.6.5 Symbol-Hunk

Anstelle eines Source-Level-Debuggers wie dem hochkomfortablen MaxonSDB kann man natürlich auch einen symbolischen Debugger wie *„MetaScope"* benutzen. Um dabei überhaupt etwas wiederzufinden, sollte man sich bei Exe- und Objektdateien sogenannte ***„Symbol-Hunks"*** erzeugen lassen. Genau dafür dient die gleichnamige Option.

Wichtig ist, daß Sie ***„Symbol-Hunk"*** schon beim Compilieren und nicht erst beim Erzeugen der Exe-Dateien anwählen, um den Compiler zu veranlassen, die notwendigen Informationen zu sammeln und aufzubewahren.

2.5.2.6.6 Asm schreiben

Wenn ***„Asm schreiben"*** angewählt ist, erzeugt der Compiler zu jedem Text, den er erfolgreich übersetzt, ein Assembler-Listing. Der Assembler-Quelltext wird in eine Datei geschrieben, deren Name wie der des C-Quelltexts anfängt und mit *„.s"* endet, also z. B. *„ballo.s"* zu *„ballo.c"*. Wenn Sie schon andere Compiler näher kennen, wundern Sie sich vielleicht darüber, daß der Compiler scheinbar einen so optimierten Assembler-Text erzeugt, daß für den Assembler nichts mehr zum Optimieren übrig bleibt. Auch werden Sie sich vielleicht fragen, warum Sie mit dem integrierten Assembler von MaxonC++ nicht selbstgeschriebene Assembler-Programme übersetzen lassen können.

In beiden Fällen handelt es sich um Fehlschlüsse, denn MaxonC++ ist etwas ungewöhnlich implementiert. Der Compiler erzeugt überhaupt keinen Assembler-Quelltext, und es gibt auch keinen Assembler.

Um diese kryptische Bemerkung etwas zu erhellen: Der Compiler erzeugt bei der Codegenerierung einen Zwischencode in einem internen binären Format, das zwar in gewisser Weise wie ein Assembler-Quelltext aufgebaut ist, sonst aber absolut nichts mit einer ASCII-Textdatei zu tun hat. Der „Assembler" ist dann ein ziemlich dummes kleines Prográmmchen, daß diesen Zwischencode in echten Maschinencode übersetzt und dabei noch ein paar kleine Optimierungen durchführt. Diese Low-Level-Optimierungen, zu denen z. B. Short Branches gehören, trägt der Assembler dann auch immer gleich im Zwischencode ein, damit er beim nächsten Pass dasselbe nicht noch einmal optimieren muß.

Es sei noch einmal darauf hingewiesen, daß dieser „Assembler" wirklich ein ziemlich dummes Programm ist, das z. B. nur eine Teilmenge der 680x0-Assemblersprache beherrscht und sich stets darauf verläßt, daß der „Quelltext" - also dieses interne Format, das ihm als Eingabe dient, fehlerfrei ist. Somit ist dieser Wannabe-Assembler absolut NICHT IN DER LAGE, NORMALE ASSEMBLER-QUELLTEXTE ZU VERARBEITEN. Und dann gibt es noch einen Programmteil, der noch viel simpler als der „Assembler" ist und den internen Zwischencode in lesbaren Assembler-Quelltext umwandelt. Diese

Funktion wird benutzt, wenn die Option ***„Asm schreiben"*** angewählt ist, und da die Datei geschrieben wird, nachdem der Pseudo-Assembler seine Arbeit getan hat, sind darin also auch schon alle Optimierungen vorgenommen. Der von MaxonC++ generierte Assembler-Source entspricht also genau dem erzeugten Code - Sie können sich den Reassembler sparen, wenn sie die Codequalität genauer unter die Lupe nehmen wollen.

2.5.2.6.7 Source in Asm

Den oben beschriebenen Quelltext kann man sich etwas übersichtlicher gestalten lassen, indem man zusätzlich die Option ***„Source in Asm"*** wählt. Dann wird nämlich der C-Quelltext an jeweils passender Stelle in den Assembler-Source als Kommentar eingefügt, so daß sich die Bedeutung der endlosen Assembler-Listings dem menschlichen Leser einfacher erschließt.

2.5.2.6.8 Optimieren

Guter Code ist mit einem gewissen Aufwand verbunden, sowohl in Bezug auf den Speicherplatzbedarf des Compilers als auch auf die Übersetzungszeit. Deshalb gibt es die wahlweise ab- oder zuschaltbare Option ***„Optimieren"***, die MaxonC++ veranlaßt, die Codeerzeugung etwas schlauer (und teurer) aufzuziehen.

Das Thema ***„Optimieren"*** ist nahezu unerschöpflich, und folglich dürfte bei jeder neuen Version von MaxonC++ das eine oder andere Feature hinzukommen. Im Moment macht der Compiler bei aktiver Optimier-Option mindestens folgendes:

- ☞ Alle anderweitig entbehrlichen Register werden für lokale Variablen benutzt. Die Entscheidung, welche Variablen in Register geladen werden sollen, fällt über gewichtetes Zählen ihres Auftretens.
- ☞ Wenn sich herausstellt, daß eine Funktion keine lokalen Variablen auf dem Stack mehr benötigt (z. B. weil alle in Register umgelegt werden konnten), wird das Einrichten eines Stack-Rahmens mittels `"LINK"` und `"UNLK"` unterdrückt.
- ☞ Der Assembler führt nicht nur einen, sondern zwei optimierende Passes durch. Damit werden z. B. auch Sprünge nach vorn zu Short-Branches optimiert.

Diverse Mini-Optimierungen, die keinen besonderen Aufwand erfordern, werden natürlich immer durchgeführt und sind nicht abschaltbar. Die Grenze zwischen „vermiedenem schlechtem Code" und einer „Optimierung" ist bekanntlich fließend.

2.5.2.6.9 Code für höhere Prozessorgenerationen

Die bewährten MC68000-Prozessoren sind in der Amiga-Welt eine aussterbende Spezies. 32Bit-Prozessoren (68020 bis 68060) und Floating Point Units (68881 und 68882) setzen sich mehr und mehr durch. Damit stehen dem Programmierer neue, leistungsfähige Maschinenbefehle zur Verfügung, die teilweise eine erstaunliche Geschwindigkeitssteigerung bringen (das ganze ist allerdings nicht so dramatisch wie in der INTEL-Welt, wo die „schnellen" 468er-DOS-PC's wegen der Zwangskompatiblität zum 8-Bit-Prozessor 8088 meist nur einen Teil ihrer Möglichkeiten nutzen können - aber lassen wir das).

Die 32-Bit-Prozessoren bieten jedenfalls insbesondere die lange vermißten Operationen für Langwort-Multiplikation und -Division und einige nette Features für die Handhabung von Arrays und Bitfeldern. Mit dem Switch ***„68020/30/40"*** veranlassen Sie den Compiler, dies auch zu nutzen. Aber Achtung: Ihr Code ist dann natürlich auf einem altem Amiga 500 wahrscheinlich nicht mehr lauffähig und schmiert an unerklärlicher Stelle plötzlich ab. Und allzu drastische Geschwindigkeitssteigerungen sollten Sie von dieser Option nicht erwarten.

Bei Programmen, die viel mit Fließkommazahlen rechnen, bringt die Anschaffung einer FPU allerdings durchaus beachtliche Performancezuwächse. Die Option ***„68881/82"*** nutzt bei der Codeerzeugung diese Möglichkeit aus.

2.5.2.6.10 Strategien für Template-Funktionen

Templates sind ein ausgesprochen nützliches und leicht verständliches neues Sprachkonstrukt, leider aber nicht ganz einfach zu implementieren. Vor allem die Funktionstemplates haben einige überraschende Eigenschaften.

Man kann im allgemeinen nicht sagen, ob die Definition eines Templates richtig oder falsch ist. Stellen Sie sich beispielsweise einmal eines der beliebten Funktionstemplates für einen Sortieralgorithmus (z. B. Quicksort) vor. Der Elementtyp des zu sortierenden Vektors ist dabei üblicherweise ein Templateargument. Wenn nun die Funktion Vergleichsoperatoren wie `"<"` oder `">="` verwendet, kann aus dem Template nur für solche Typen, auf denen diese Operatoren definiert sind, eine Funktion generiert werden. Der Programmierer kann dem abhelfen, indem er die benötigten Funktionen für andere Datentypen „von Hand" implementiert. Da man bei einem größeren Projekt aber meist mehr als eine Übersetzungseinheit hat, kann der Compiler nicht wissen, welche der nicht erzeugbaren und trotzdem benötigten Funktionen irgendwo im Gesamtprojekt bereits fertig vorliegen.

Aus dieser Misere behilft sich MaxonC^{++} mit zwei verschiedenen Strategien, aus denen Sie mit der Option ***„Alle Templates"*** aus dem Compiler-Dialogfenster auswählen können:

Wenn die Option angewählt ist, unterstellt der Compiler, daß der Programmierer sich nicht die Mühe gemacht hat, irgendwelche Templatefunktionen von Hand zu programmieren. Deshalb erzeugt er alle Templatefunktionen, die in der vorliegenden Übersetzungseinheit benötigt werden und dort sonst nicht definiert sind. Tritt dabei ein Fehler auf, so wird er gemeldet. Funktionen, die dabei überflüssigerweise erzeugt wurden, werden später vom Linker 'rausgeschmissen.

Eine andere Strategie wird verfolgt, wenn die Option ***„Alle Templates"*** nicht angewählt ist. Auch dann werden „auf Verdacht" alle in Frage kommenden Templatefunktionen generiert. Wenn dabei in einer Funktion aber ein Fehler gefunden wird, wird der Versuch verworfen. Es wird keinerlei Code für die fehlerhafte Funktion erzeugt, dafür wird der Programmierer aber auch nicht mit der Fehlermeldung belästigt. Der Compiler nimmt hier also an, daß alle nicht fehlerfrei erzeugbaren Templatefunktionen irgendwo im Gesamtprojekt sinnreich definiert wurden.

Während der Programmentwicklung rate ich Ihnen dringend, die fragliche Option zunächst einmal eingeschaltet zu lassen. Dies ist nämlich Voraussetzung dafür, daß Sie überhaupt eine aussagekräfti-

ge Meldung über Fehler in einer Templatefunktion erhalten (andernfalls teilt Ihnen lediglich der Linker lapidar mit, daß die Funktion offenbar nicht erzeugt werden konnte, und zwar ohne Angabe von Gründen oder Fehlerstellen). Wenn Sie aber mit fertigen Template-Bibliotheken arbeiten oder ein bereits vorhandenes Programm portieren möchten, kann es irgendwann notwendig werden, die Zwangsgenerierung aller Templatefunktionen abzustellen.

Falls Sie sich jetzt fragen sollten, wieso Sie sich mit derartigen Strategie-Auswahlen herumplagen müssen, so verrate ich Ihnen jetzt, wie eine korrekte Implementierung von Funktionstemplates genaugenommen vorgehen müßte:

- Bei der ersten Übersetzung werden überhaupt keine Templates erzeugt.
- Der Linker versucht, das Projekt zusammenzubinden, und erstellt dabei eine Liste aller undefinierten Funktionen.
- Im finalen Compilerdurchlauf werden alle Module, die irgendwie in dem Ruch stehen, eine hilfreiche Templatedefinition zu enthalten, erneut compiliert. Dabei kann anhand der vom Linker erstellten Liste sicher entschieden werden, welche Funktionen generiert werden müssen.

Eine solche Implementierung wäre nicht nur sehr aufwendig, sondern auch enorm langsam, da jedesmal diverse Module doppelt compiliert werden müßten. Deshalb greift MaxonC++ auf die Methode mit der Strategiewahl durch den Programmierer zurück.

2.5.2.6.11 Exceptions

Wie in Kapitel 8.5 des Tutorials dargestellt, hat es manchmal Vorteile, wenn man ein "klassisches" C++-Programm, das kein Exception Handling benutzt, in einem speziellen Compilermodus übersetzen läßt. Dann werden die mit der Ausnahmebehandlung zusammenhängenden Schlüsselwörter (`"catch"`, `"throw"` und `"try"`) nicht mehr erkannt, und im erzeugten Code wird nicht über notwendige Destruktoraufrufe Buch geführt. Wenn der Schalter "***Exceptions***" ausgeschaltet ist, ist also keine Ausnahmebehandlung möglich. Der Zugewinn an Geschwindigkeit und Kürze ist aber meist gering.

2.5.2.7 Optionale Warnungen und Meldungen

Diverse Sprachkonstrukte sind zwar an sich syntaktisch wie semantisch korrekt, aber trotzdem etwas obskur und möglicherweise Folge eines Versehens des Programmierers. Deshalb gibt es elf Warnungen, die man je nach Situation und persönlichem Programmierstil einzeln ein- und ausschalten kann. In MaxonC++ dienen dazu die Switches in der rechten Spalte des Compiler-Dialogs.

Im Einzelnen gibt es die folgenden Warnungen:

- ***„Kommentare"***: Bei verschachtelten Kommentaren, wie `"/* Test /* Nahu? */"` wird gewarnt. Hier liegt der Verdacht nahe, daß am Ende eines Kommentars einfach das `"/*"` vergessen wurde.
- ***„Prototypen"***: Im C-Modus wird gewarnt, wenn eine Funktion aufgerufen wird, zu der kein Prototyp deklariert wurde. In C++ ist das bekanntlich sowieso ein Fehler.

- *„'return' fehlt"*: Wenn in einer Funktion, deren Ergebnistyp nicht `"void"` ist, nirgendwo ein `"return"` auftaucht, ist sicher etwas faul. Deshalb sollte diese Warning immer eingeschaltet sein.

- *„Leeranweisung"*: Eine Anweisung wie `"42;"` hat keinerlei Wirkung und war folglich wahrscheinlich nicht so gemeint. Ein besonders beliebter Fehler ist auch der Aufruf einer parameterlosen Funktion ohne Argumentliste, etwa `"test;"` statt `"test();"`.

- *„Var. unbenutzt"*: Hier wird gewarnt, wenn eine Variable im automatischen Speicher deklariert, aber nie benutzt wird (dann ist sie wohl überflüssig), oder wenn sie benutzt, aber nirgendwo initialisiert wird (das ist wohl ein Fehler).

- *„K&R-Parameter"*: Das „K&R" steht für „Kernighan & Ritchie" und damit für den Vor-ANSI-Standard, der übrigens keiner war. Mit dieser Option kann man sich bei Parameterlisten im alten Stil eine Warnung ausgeben lassen.

- *„#pragma"*: Es ist natürlich Sinn und Zweck eines `"#pragma"`, daß der Compiler darin eben keine Fehler meldet und alles ignoriert, was er nicht kennt. Wenn Sie aber nicht bloß fremde Quelltexte übernehmen, sondern mit MaxonC^{++} selbst entwickeln, ist es sicher sinnvoll, wenn Sie sich auf unbekannte Pragma's hinweisen lassen - schließlich könnte es sich dabei ja auch ganz einfach um einen Tippfehler handeln.

- *„Temporarys"*: Laut C^{++}-2.0-Standard ist es ein Fehler, wenn eine Referenz auf einen nichtkonstanten Typ mit einem Nicht-L-Wert initialisiert wird und dabei ein temporäres Objekt eingeführt werden muß. Da diese Regel im 1.0-Standard noch nicht existierte, gibt es dafür keine „echte" Fehlermeldung, sondern eine abschaltbare Warnung.

Ein Beispiel:

```
void dup(int &ir)
{ ir *= 2; }

void main()
{ int i;    dup(i);        // OK
long l;  dup(l);           // VORSICHT! }
```

Da hier `"l"` von `"int"` nach `"long"` konvertiert werden muß (in MaxonC^{++} übrigens eine leere Operation, denn intern sind beides 32Bit-Zahlen), wird ein temporäres Objekt eingeführt, und der Programmierer erlebt höchstwahrscheinlich eine Überraschung, denn beim zweiten `"dup"`-Aufruf gibt es keinen Seiteneffekt auf das Argument. Die neue C^{++}-Regel ist also ausgesprochen sinnvoll.

- *„Unsichere Konv."*: Diese Option warnt bei impliziten Typumwandlungen (also ohne Cast) von ganzzahligen in Fließkommatypen.

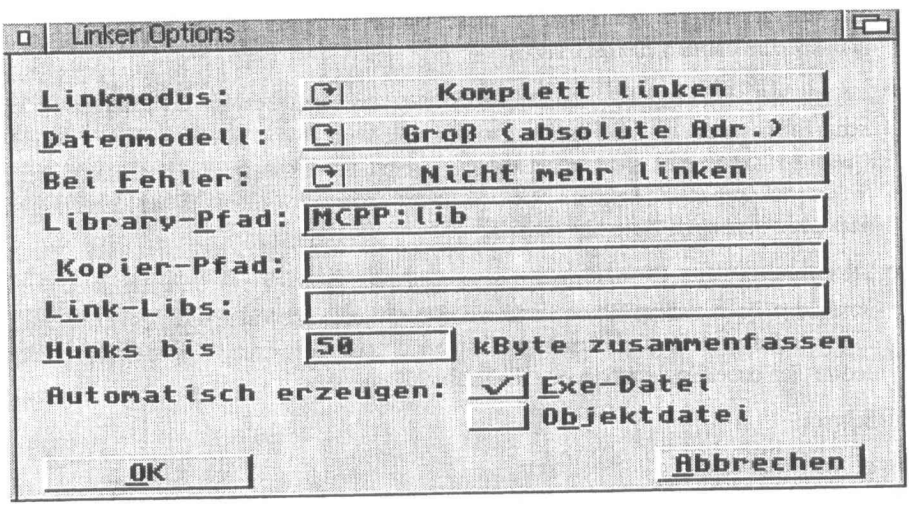
- *„Verdächtiges =="*: Dabei handelt es sich um das beliebte Konstrukt `"if (a=b)..."` - korrekt und bei vielen Programmierern beliebt, aber auch ein typischer Anfängerfehler statt `"if (a==b)"`. Wenn die entsprechende Option aktiv ist, wird jeweils Warnung ausgegeben, wenn

ein `"="` auf diese Weise in einem logischen Ausdruck auftritt, also als Bedingung in `"if"`, `"while"`, `"for"` oder `"do"` sowie als Operand von `"||"`, `"&&"` oder `"!"`.

- ***„Makro-Parameter"***: Wenn ein Argument für ein Preprozessor-Makro sich über mehr als 8 Zeilen oder 200 Token erstreckt, liegt der Verdacht nahe, daß hier schlicht eine schließende Klammer vergessen wurde. Damit der Programmierer (das sind Sie) dann nicht endlos suchen muß, gibt es diese optionale Warnung.

2.5.3 Der Linker-Requester

Mit dem Menüpunkt **„Linker"** aus dem **„Optionen"-Menü** öffnen Sie den Linker-Dialog. Hier finden Sie diverse Optionen zur Steuerung des Linkers.



2.5.3.1 Datenmodelle

Die Umschaltung zwischen ***„kleinem"*** und ***„großem" Datenmodell*** hat sich eher zufällig in den Linker-Dialog verirrt, da dies eigentlich eher etwas mit der Codeerzeugung und folglich mit dem Compiler zu tun hat.

Zuerst einmal die Frage: Was hat es eigentlich mit diesen Datenmodellen auf sich?

Wenn im vom Compiler generierten Code auf globale, statische Daten zugegriffen werden soll, geschieht dies normalerweise über absolute Adressen. Die Daten selbst können dann irgendwo im Speicher liegen und über beliebig viele Daten-"Hunks" verteilt sein. Auch die Gesamtgröße dieser Daten ist nur durch die Speichergröße des Rechners begrenzt. Diese kanonische Art der Adressierung bezeichnet man in diesem Zusammenhang als „großes" Datenmodell („Large Data").

Das klappt zwar recht gut und unterliegt auch keinerlei Beschränkungen, aber wie so oft gibt es auch hier Leute, die daran etwas zu mosern haben. Zum Beispiel benötigt eine absolute Adresse im

Code immer 32 Bit, also vier Bytes. Das bläht nicht nur die Codegröße auf, sondern kostet auch ein wenig mehr an Laufzeit. Ferner müssen alle diese Adressen korrigiert werden, wenn das Programm in den Speicher geladen wird, was in der Dateigröße einen gewissen Overhead bedeutet.

Deshalb ist ein anderer Ansatz, das sogenannte „kleine" Datenmodell, in gewisser Hinsicht schlauer. Hier wird ein Prozessorregister, nämlich das Adressregister A4, als Zeiger auf die statischen Daten des Programms benutzt. Die Daten werden jetzt mit einem 16 Bit-Offset relativ zu diesem Datenregister adressiert, was jeweils zwei Bytes Code und ein Nichtigstel an Laufzeit spart. Aber jetzt kommen die Nachteile von „Small Data":

- Mit 16 Bit kann man nur 65536 Bytes adressieren - also ist dies die Obergrenze für die globalen Daten des Programms.
- Alle Datenhunks des Programms müssen in einem einzigen Small-Data-Hunk zusammengefaßt werden. Insbesondere unterscheidet man dann nicht mehr zwischen initialisierten Daten („DATA"-Hunk) und den platzsparend abgespeicherten uninitialisierten Daten („BSS"-Hunk).
- Bei der Codegenerierung kann das Register A4 nicht für Variablen oder Zwischenergebnisse genutzt werden.

Welche Konsequenz soll man aus diesen Vor- und Nachteilen ziehen?

So ganz allgemein wohl gar keine. Im Zweifelsfall sollten Sie einfach ausprobieren, mit welchem Datenmodell ihr Programm sich besser macht.

Rein praktisch wird das Datenmodell in *MCPP* mit dem entsprechenden Gadget im „Linker"-Dialog und beim „mcppc" mit der Option `"-gd"` bzw. `"-gD"` eingestellt.

Noch eine Anmerkung zum Schluß: Ich erwähnte oben, daß MaxonC++ bei „large data" das Register A4 munter benutzt. Wie paßt das mit der Aussage zusammen, daß man small- und large-data-Code beliebig mischen kann? Nun, A4 wird auch bei „großem" Datenmodell nur in solchen Funktionen getrasht, die selbst keine weiteren Funktionen aufrufen - und folglich insbesondere keine Funktionen mit A4-relativen Daten. Natürlich wird in solchen Funktionen A4 auch gegebenenfalls auf den Stack gerettet und am Ende der Funktion wieder hergestellt. Damit merkt die gesamte „Umwelt" nichts davon, wenn in einer large-data-Funktion A4 mißbraucht wird.

2.5.3.2 Linken oder nicht Linken - das ist nier die Frage

Auch unter MaxonC++ haben Sie die Möglichkeit, d[illegible]inken ganz oder teilweise zu unterlassen, wenn auch nicht mit ganz so vielen Optionen wi[illegible] mcppc. Dazu dient das Gadget ***„Linkermodus"*** im Linker-Requester. Die vier Einstellungen:

- ***„Komplett linken":*** Das übersetzte Programm wird ganz normal mit den Bibliotheksroutinen zu einem ausführbaren Programm zusammengebunden - oder zumindest wird es versucht.
- ***„Nicht linken":*** Es werden keinerlei Bibliotheks- oder sonstige Module hinzugeladen. Der Linker meldet auch keine Fehler, sondern wird überhaupt nicht erst gestartet. Dieser Modus ist

recht angenehm, wenn man an einem Executable nicht interessiert ist und lediglich eine Objektdatei haben möchte.

☞ ***„Ohne Startup-Code“:*** Vorsicht, eine Einstellung für Spezialisten - wenn man damit nicht umgehen kann, stürzen die erzeugten Programme gnadenlos ab! Näheres über diese Option dazu finden Sie im Linker-Kapitel, und zwar in Abschnitt 4.6.

☞ ***„Library erzeugen“:*** Bei dieser Einstellung wird nach Möglichkeit versucht, das Programm zu linken. Vor allem wird aber während der Codeerzeugung für jede Funktion und jede globale Variable ein eigener „Hunk“ erzeugt (wenn Sie nicht wissen, was ein Hunk ist, macht das nix. Sie können aber auch einfach im Index nachschlagen!). Die Folge: Der Linker kann nicht benutzte Funktionen einfach herausschmeißen und so die Codegröße verringern. Dies ist praktisch, wenn man eine Bibliothek programmieren möchte, kostet ansonsten aber ein bißchen was: Funktionen einer Übersetzungseinheit können z. B. identische Zeichenketten nicht mehr gemeinsam benutzen und sich vor allem gegenseitig nicht mehr PC-relativ aufrufen.

2.5.3.3 Verhalten bei Fehler

Die 1.x-Version von *MCPP* versuchte immer, das Programm zu linken, auch dann, wenn bei der Übersetzung Fehler entdeckt wurden. Mit dem Schalter ***„Bei Fehler“*** können Sie dieses vielerseits unbeliebte Verhalten unterbinden: Wird bei der Übersetzung ein Fehler (gilt nicht für Warnungen!) gefunden, wird der Linkvorgang schlicht unterlassen.

2.5.3.4 Bibliotheken

Der Linker von MaxonC++ muß natürlich auch irgendwie wissen, woher er die Standard-Bibliotheken laden muß. Deshalb trägt man die entsprechenden Pfade im ***„Lib-Pfade“***-Gadget des Linker-Dialogs ein. Genau wie bei den Includedateien (siehe 2.5.2.1) kann man auch hier bis zu zwei verschiedene Pfade angeben und sich Dateien dabei automatisch umkopieren lassen. Von sich aus versucht der Linker immer nur, undefinierte Symbole im *„Logfile“* zu suchen und anschließend die entsprechenden Bibliotheksmodule zu laden. Wenn Sie eine Linker-Library, etwa die allseits beliebte *„amiga.lib“*, einbinden wollen, müssen Sie deren Namen (ohne *„.lib“*) im Gadget ***„Link-Libs“*** eintragen. Da es nur ein solches Gadget gibt, können Sie darin auch mehrere solcher Bibliotheken durch Leerzeichen getrennt eintragen, z. B. `"amiga debug murx"`. Diese Bibliotheksnamen können wahlweise mit einem vollständigen Pfad oder ganz ohne Pfad angegeben werden. In letzterem Fall sucht der Linker sie wie die „normalen“ Bibliotheksmodule im dafür spezifizierten Verzeichnis.

2.5.3.5 Zusammenfassen von Hunks

Jedes Modul eines Programms besitzt normalerweise einen Code-Hunk und außerdem noch bis zu drei verschiedenen Datenhunks. Bei einem Programm, das aus mehreren Übersetzungseinheiten besteht, kommt da schon so einiges zusammen, einmal ganz zu schweigen von den Bibliotheksmodulen. Damit das mit der Hunkanzahl nicht Überhand nimmt, ist es üblich, gleichartige Hunks zu einem einzigen zusammenzufassen.

Je weniger Hunks ein Programm besitzt, desto kleiner ist die ausführbare Programmdatei, denn jeder Hunk benötigt einen Overhead von ein paar Bytes. Andererseits ist es aber auch nicht schlau,

grundsätzlich alle jeweils gleichartigen Hunks zusammenzufassen, denn ein 250 kByte großer Hunk benötigt nun einmal einen ebenso großen, zusammenhängenden Speicherbereich. Bei Rechnern, die knapp an RAM sind, kommt es durchaus vor, daß der Speicher so fragmentiert ist, daß zwar durchaus noch fünf Blöcke zu je 50 kByte, aber kein einziger zusammenhängender 250k-Block frei ist - ein Programm mit einem unangemessen großen Hunk könnte also nicht mehr geladen werden. Deshalb kann man eine Höchstgrenze wählen, die der Linker beim Kombinieren von Hunks nicht überschreiten soll. Der entsprechende Betrag ist (in kByte) im entsprechenden Gadget des Linker-Requesters einzutragen. Wenn Sie z. B. aus irgendwelchen Gründen wollen, daß überhaupt keine Hunks zusammengefaßt werden, setzen Sie den Wert einfach auf Null.

In einem Fall nimmt sich der Linker aber auf jeden Fall eine gewisse Freiheit heraus: Bibliotheksmodule werden immer zu Blöcken von 10 kByte kombiniert, auch wenn Sie einen kleineren Wert vorgegeben haben. Das liegt daran, daß die automatisch eingebundenen Bibliotheksfunktionen naturgemäß aus etlichen kleinen Hunks bestehen und es einfach Wahnsinn ist, jeden davon im ausführbaren Programm einzeln auftauchen zu lassen.

2.5.3.6 Schreiben von Dateien

Einer der Vorzüge einer integrierten Umgebung wie *MCPP* besteht darin, daß sich so ziemlich alles im RAM und ohne Disketten- bzw. Festplattenzugriffe abspielen kann: Die Quelltexte werden aus dem RAM gelesen und das ausführbare Programm ebenda abgelegt.

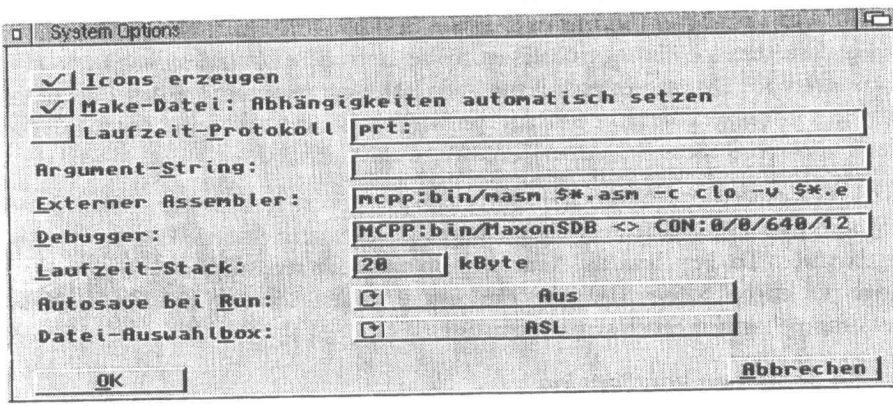
Nun gibt es aber bestimmte Gründe dafür, sich stets eine aktuelle ausführbare Programmdatei schreiben zu lassen.

Deshalb gibt es im „Linker"-Dialog ein Symbol, das Sie aktivieren können, wenn nach jeder erfolgreichen Übersetzung eine Exe-Datei (natürlich einschließlich der zugehörigen *„.mdbg"-Datei*, falls die entsprechende Compileroption angewählt ist) automatisch erzeugt werden soll.

Mit einem weiteren Gadget können Sie sich außerdem Objektdateien automatisch erzeugen lassen. Im Make-Modus geschieht das natürlich sowieso, so daß dieses Gadget dann keine Bedeutung hat.

2.5.4 MCPP-Systemeinstellungen

Mit dem dritten Dialogfenster, das Sie mit **„System"** aus dem **„Optionen"-Menü** öffnen, können diverse Einstellungen der MCPP-Umgebung vorgenommen werden.



2.5.4.1 Erzeugen von Icons

„Icons erzeugen" legt fest, ob MaxonC++ beim Schreiben von ausführbaren Dateien, Makefiles und Debugger-Dateien jeweils ein Icon erzeugen soll.

2.5.4.2 Abhängigkeiten in Make-Dateien

Die integrierte Projektverwaltung entscheidet souverän, wann welches Modul eines größeren Programms neu compiliert werden muß. Dazu muß sie aber wissen, von welchen Quelltextdateien ein Objekt abhängig ist. Diese Abhängigkeiten kann der Benutzer durchaus von Hand setzen - bequemer und sicherer ist es allerdings, wenn man diese Aufgabe MCPP überläßt.

Wenn sie das Symbol *„Make-Datei: Abhängigkeiten automatisch setzen"* angewählt haben, paßt *MCPP* beim Compilieren auf, welche Includedateien bei den einzelnen Objektdateien gelesen werden, und trägt diese dann als Abhängigkeit in die Make-Datei ein.

2.5.4.3 Protokollieren von Ein- und Ausgaben

Wenn Sie die Option *„Protokoll"* einschalten, werden sowohl die Ausgaben des Programms als auch die Eingaben des Benutzers in eine Datei geschrieben. Der Dateiname ist im Textgadget einzutragen, wobei `"prt:"`, also der Drucker, als Defaultwert vorgegeben wird.

2.5.4.4 Argumente aus der imaginären Kommandozeile

Beim Start von der Shell kann man Programmen Argumente übergeben, die in C mit den Parametern `"argc"` und `"argv"` ausgewertet werden können. Auch beim Programmstart aus der Entwicklungsumgebung von MaxonC++ kann man diese Art der Parameterübergabe simulieren, indem man sie im Textgadget `"Argumente"` eingibt. Diese Argumentzeile wird dann an aus *MCPP* gestartete Programme übergeben. Beachten Sie bitte, daß der erste Teil des Strings als Programmname interpretiert und in `"argv[0]"` übergeben wird.

2.5.4.5 Einbindung des Maxon Assemblers

Seit Version 3 beherrscht das integrierte Make auch sogenannte Regeln, die möglichenfalls automatisch angewendet werden, wenn für eine Objektdatei keine Compileranweisung vorhanden ist. Zwei dieser Regeln sind sogar schon „eingebaut": Zum einen die `".c.o"`-Regel, die besagt, daß eine auf `".o"` endende Datei, zu der eine `".c"`-Datei gleichen Namens existiert, als C-Programm compiliert werden kann, und zum anderen die `".asm.o"`-Regel für Assemblersourcen. Diese Regel können Sie über das Eingabefeld ***„Externer Assembler"*** frei bestimmen.

Default ist

```
mcpp:bin/masm $*.asm -c cl -v $*.err
```

Das `"$*"` wird (wie immer bei Make-Regeln) durch den endungslosen Dateinamen ersetzt. Also wird für jede auf `".o"` endende Objektdatei, für die es im Makefile keine Compileranweisung, dafür aber auf der Platte eine passende `".asm"`-Datei gibt, automatisch ein solcher Maxon Assembler-Aufruf erzeugt.

Diese Regel wird auch benutzt, wenn mit dem Menüpunt **„Text und Objekt aufnehmen"** eine auf `".asm"` endende Quelltextdatei in das Makefile aufgenommen wird. Auch in diesem Fall erhalten Sie also automatisch einen passenden Assembleraufruf.

Vielleicht fällt Ihnen die `"-v"`-Option auf. Der Maxon Assembler ist inzwischen so weit in das System integriert, daß *MCPP* die MaxonAsm-Fehlerdateien interpretieren kann (und genau diese werden durch `"-v"` erzeugt). Wenn MCPP ein Assemblermodul auf diese weise „extern" übersetzen läßt, liest es anschließend diese Fehlerdatei ein und zeigt die Assemblerfehler im normalen Compiler-Fenster an. Prima, nicht?

2.5.4.6 Debugger

Der Menüpunkt **„Debugger"** ruft bekanntlich den MaxonSDB auf (siehe auch 2.4.3). Damit *MCPP* auch weiß, was dabei zu tun ist, gibt es das Gadget ***„Debugger"*** im System-Einstellfenster. Hier ist die Shell-Anweisung anzugeben, die von *MCPP* ausgeführt werden soll (bis auf den Dateinamen, denn den hängt *MCPP* einschließlich des Argumentstrings immer automatisch an). Default ist

```
mcpp:bin/MaxonSDB <> CON:0/0/640/120/MaxonSDB
```

Wie Sie sehen, wird hier gleichzeitig das Fenster für die Ein- und Ausgaben des debuggten Programms festgelegt.

2.5.4.7 Stackgröße

Wenn ein Programm aus *MCPP* gestartet wird, braucht es natürlich auch einen Stack, und im Zweifelsfall nicht zu wenig davon. Die gewünschte Größe kann ebenfalls im „System"-Dialog eingestellt werden.

2.5.4.8 Autosave

Wenn die ***„Autosave"***-Option vom *MCPP* eingeschaltet ist, werden beim Start eines Programms aus der integrierten Umgebung automatisch alle Editor-Texte abgespeichert, sofern dies noch nicht

geschehen ist - bestimmt eine sinnvolle Maßnahme, zumal man in C stets ohne Netz und doppelten Boden programmiert. Dieser Autosave sollte nicht mit dem zeitabhängigen Autosave des *Edward* verwechselt werden.

Es gibt hier sogar gleich drei verschiedene Einstellungen:

☞ ***Aus:*** Der MCPP-Autosave unterbleibt.

☞ ***Texte sichern:*** Die Texte im Editor werden nötigenfalls abgespeichert.

☞ ***Texte und Session:*** Zusätzlich werden die *„.SESSION"-Dateien* von *Edward* und *MCPP* geschrieben. Dadurch haben Sie nach einem Absturz und erneutem Start von *MCPP* wieder alles so vor sich, wie es gewesen war.

2.5.4.9 Dateiauswahlbox

Der RCT-Filerequester hat seine Fans (mich eingeschlossen), aber der Commodore sieht es gar nicht gern, wenn man gegen seine Richtlinien verstößt. Deshalb kann zwischen ***RCT-*** und ***ASL-Dateiauswahlbox*** frei gewählt werden.

2.5.5 Abspeichern der Konfigurationsdatei

2.5.5.1 Allgemeines

Wenn im aktuellen Verzeichnis keine Datei *„mcpp.SESSION"* gefunden wird, liest *MCPP* die Einstellungen aus der Datei *„s:mcpp.CONFIG"* (siehe auch Abschnitt 1.3.3), und wenn es die auch nicht gibt, werden eben die internen Defaults verwendet. Mit dem Menüpunkt **„Optionen sichern"** werden alle aktuellen Einstellungen von *MCPP* (ausgenommen natürlich der Editor-Zustand) in *„s:mcpp.CONFIG"* abgespeichert.

An dieser Stelle soll noch darauf eingegangen werden, wie diese Konfigurations- und Session-Dateien eigentlich aufgebaut sind. Zwar werden Sie darin kaum einmal etwas von Hand ändern, aber Sie müssen es wissen, wenn Sie über den ARexx-Befehl `"setopt"` MCPP-Einstellungen ändern (siehe 2.6.3.6)

Die Konfigurationsdateien sind Texte mit einer beliebigen Folge von Optionen, die jeweils mit einem `"-"` beginnen. Dabei entsprechen die Compiler- und Linker-Einstellungen jeweils exakt denen von „mcppc": `"-gO"` schaltet die Optimierung ein, `"-wA"` warnt bei verdächtigen Zuweisungen in Bedingungen usw., etc. und so fort.

Für alle anderen Einstellungen von *MCPP* gibt es spezielle Optionen, die alle mit `"-x"` beginnen und vom *mcppc* naturgemäß nicht verstanden werden.

2.5.5.2 Der Editor

Zunächst ist da ein Eintrag `-x"Edward:Edward"`

Wie Sie sich vielleicht denken können, entnimmt *MCPP* dieser Angabe, woher der Editor *Edward* geladen werden soll. Falls Edward in Ihrem System nicht unter diesem Pfad und Namen zu finden

sein sollte, können Sie diesen Eintrag ändern. Sie sollten aber nicht unbedingt versuchen, auf diese Weise einen anderen Editor als *Edward* einzubinden, denn nach derzeitigem Stand beherrscht kein fremder Editor das kompilzierte Message-Protokoll zwischen *MCPP* und *Edward*. Alternative Editoren können ausschließlich über ARexx (Kapitel 2.6) integriert werden.

2.5.5.3 Fensterdaten

Am Anfang der Konfigurationsdatei finden Sie gewöhnlich eine ganze Reihe von Einträgen, die mit `"-Xn"` beginnen (wobei `n` eine Zahl zwischen `1` und `9` ist). Diese Daten legen Position und Größe der Windows von MCPP fest und sollten nie manuell verändert werden.

2.5.5.4 Switches und Schalter

Die Optionen `"-X10"` bis `"-X18"` repräsentieren diverse Schalterstellungen der integrierten MCPP-Umgebung. Es folgen jeweils stets ein Komma und die Schalterstellung - meist `"0"` oder `"1"`, manchmal aber auch höhere Nummern.

Und das bedeuten diese Einträge:

`-X10:` Exe-Datei automatisch erzeugen?
`-X11:` Objektdatei automatisch erzeugen?
`-X12:` Bei Compilerfehlern Linken unterlassen?
`-X13:` Icons erzeugen?
`-X14:` Protokolldatei schreiben?
`-X15:` Autosave (0 = Aus, 1 = Texte sichern, 2 = Texte + Session)
`-X16:` Größe des Laufzeitstacks
`-X17:` Linkermodus (0 = Nicht linken, 1 = Komplett linken, 2 = Ohne Startupcode, 3 = Library erzeugen)
`-X18:` Dateiauswahlbox (0 = RCT, 1 = ASL)

2.5.5.5 Stringoptionen

Die Einträge `"-X20"` bis `"-X23"` geben den Inhalt von vier Texteingabefeldern wieder und erwarten folgerichtig als Argument eine `"so"` eingeschlossene Zeichenkette:

`-X20:` Name der Protokolldatei
`-X21:` Argumentstring bei Programmstart aus der Umgebung
`-X22:` Default-Assembleraufruf
`-X23:` Debuggeraufruf

2.6 Die ARexx-Schnittstelle

2.6.1 Grundlagen

Die Programmiersprache Rexx ist in ihrer Amiga-Portierung „ARexx" zu beachtlicher Popularität gelangt, obwohl Rexx im Wirklichkeit doch eigentlich eher eine Script-Sprache und für „echte" Programme ziemlich unbrauchbar ist. Der Grund für diesen phänomenalen Erfolg ist (einmal abgesehen davon, daß ARexx ab Workbench 2.0 zum Lieferumfang jedes Amigas gehört), daß ARexx sich in hervorragender Weise dazu eignet, Programme „von außen" zu steuern und so zum einen komplexe Abläufe zu automatisieren und zum anderen wildfremde Programme zu einem System zu kombinieren.

Voraussetzung dafür ist, daß die Programme einen sogenannten ARexx-Port besitzen. Ein ARexx-Programm kann Anweisungen an eine solche Software-Schnittstelle schicken, und wenn das entsprechende Programm über einen geeigneten Befehlssatz verfügt, kann der Rexx-Programmierer Programmfunktionen mit einem ARexx-Script fernsteuern.

Der Editor *„Edward"* besitzt eine solche Schnittstelle, und da *Edward* Anweisungen an *MCPP* weitergeben kann, läßt sich die Compiler-Umgebung natürlich über diesen Umweg steuern.

Ein Beispiel: Das ARexx-Script

```
ADDRESS "EDWARD" 'SendAppComm "compile"'
```

veranlaßt *MCPP*, den im Editor befindlichen Quelltext zu übersetzen. Voraussetzung dafür ist natürlich, daß MaxonC++ zuvor gestartet wurde. Mit der ARexx-Anweisung `"ADDRESS"` legt man übrigens fest, wie der Port heißt, mit dem das Script kommunizieren soll.

Es bleibt noch die Frage offen, was man macht, wenn man *MCPP* ohne *Edward* benutzen und fernsteuern möchte - z. B. weil man einen anderen Editor bevorzugt (nahezu völlig unverständlich!) und diesen über ARexx in das MaxonC++ System integrieren möchte.

2.6.2 Der ARexx-Port

Seit Version 3 besitzt MCPP einen eigenen ARexx-Port. In Abschnitt 2.1 wurde bereits angedeutet, daß *MCPP* dafür vom CLI bzw. der Shell aus mit der Option `"-r"` zu starten ist, z. B. `MCPP -r`

In diesem Fall wird ein Port mit dem Namen `"MCPP"` eingerichtet. Falls ein solcher schon existiert (kommt insbesondere vor, wenn MaxonC++ mehrfach gestartet wird), werden so lange `"MCPP1"`, `"MCPP2"` usw. ausprobiert, bis ein noch nicht belegter Name gefunden wird. Alternativ können Sie auch hinter dem `"-r"` einen beliebigen Portnamen vorgeben, z. B.

```
MCPP -r "Bla Fasel Sülz!"
```

Generell werden solche Portnamen in Großbuchstaben umgewandelt. Auch hier werden systemweit eindeutige Namen sichergestellt und ggf. `"BLA FASEL SÜLZ!1"` usw. ausprobiert.

In jeden Fall startet nun *MCPP*, meldet sich und gibt zur Kontrolle den Portnamen aus. *Edward* wird nicht geladen, *MCPP* kommt im Rexx-Modus völlig ohne ihn aus.

Wenn Sie den Default-Namen verwenden, können Sie Ihr Script also locker-flockig mit `ADDRESS "MCPP"` einleiten. Welche Anweisungen Ihnen dann zur Verfügung stehen, wird in den folgenden Abschnitten dargestellt. Zuvor sollten Sie aber noch erfahren, wie Sie das MaxonC^{++} wieder loswerden, ohne Ihren Rechner booten zu müssen. Zum einen gibt es dafür die Anweisung `"QUIT"`. Das folgende kurze ARexx-Script entfernt also *MCPP* aus dem System:

```
Address "MCPP" quit
```

An dieser Stelle sehen Sie auch, daß Groß-/Kleinschreibung von Befehlen in ARexx generell egal ist.

Andererseits reagiert *MCPP* im Rexx-Modus auch auf die üblichen Abbruchsignale. Wenn Sie *MCPP* aus der Shell ohne `"run"` gestartet haben, genügt also die Tastenkombination <Ctrl>+<C>. Falls Sie `"run"` verwendet haben, können Sie mit der Shell-Anweiung `"status"` die Prozessnummer herausfinden und diesen Prozeß dann mit der `"break"`-Anweisung abmurxen.

2.6.3 Die ARexx-Befehle

2.6.3.1 Schließen, öffnen und Beenden

Wie bereits in 2.6.2 erläutert, kann *MCPP* durch die Anweisung `quit` beendet werden. `shutdown` schließt alle Fenster, die *MCPP* gerade geöffnet hat, z. B. weil das ganze System Iconifizert werden soll. Anschließend können mit `reopen` alle zuvor geöffneten Fenster erneut geöffnet werden.

2.6.3.2 Übersetzen, Starten und Ähnliches

Die ARexx-Befehle von *MCPP* entsprechen im wesentlichen den Anweisungen, mit denen normalerweise *Edward* den Compiler steuert (also über `"SendAppComm"`) und die folglich auch in der *„.DEF"*-Datei verwendet werden. Allerdings haben einige Anweisungen hier Argumente, die zum Teil aber optional sind. So auch bei der wichtigsten Anweisung überhaupt: `"compile"`. `"compile hello.c"` veranlaßt *MCPP*, das gewohnte Compiler-Fenster zu öffnen und die Quelldatei *„hello.c"* zu übersetzen. ARexx-Neulinge sollten beachten, daß die Anführungszeichen in diesem Fall notwendig sind, damit ARexx `"compile"` und `"hello.c"` als Teile einer zusammengehörenden Anweisung versteht und vollständig an *MCPP* sendet.

`"compile"` ohne Dateinamen bewirkt die Compilation einer Datei namens *„default.c"* - falls vorhanden. Im Make-Modus (auch den gibt es natürlich im Rexx-Modus) werden die Dateinamen allerdings generell ignoriert und es wird stur nach der Make-Datei gearbeitet. Die Anweisung `run` startet ein übersetztes Programm, falls ein solches vorliegt. Im Gegensatz zur „normalen" Entwicklungsumgebung wird dabei nicht automatisch compiliert, und es wird erst recht nicht geprüft, ob eine erneute Übersetzung (z. B. wegen Veränderung von Quelltexten) nötig wäre.

Ein mit `run` gestartetes Programm kann mit der Anweisung `break` brutal abgebrochen werden. debug startet in gewohnter Weise den Debugger.

2.6.3.3 Requester und Informationen

Auch unter ARexx müssen Sie nicht darauf verzichten, die zahlreichen Optionen und Einstellmöglichkeiten des Systems über Dialogfenster zu wählen: **opsys** öffnet das System-Dialogfenster, **opcomp** öffnet das Dielogfenster für die Compiler-Optionen und **oplink** öffnet den Linker-Requester. Das Info-Fenster können Sie sich mit der Anweisung **info** ansehen, und nach einen Compilerdurchlauf läßt sich das bereits geschlossene Compiler-Fenster (in dem eventuell Fehlermeldungen stehen) mit der Anweisung **errors** erneut öffnen.

2.6.3.4 Make

Die Projektverwaltung steht auch im Rexx-Modus zur Verfügung. **"loadmk <Dateiname>"** lädt die angegebene Make-Datei. Wenn der Dateiname weggelassen wird, öffnet sich das Dateiauswahlfenster, und der Benutzer kann selbst eine Datei auswählen. Dagegen wird bei **newmk** stets eine vom Anwender zu benennende neue Projektdatei angelegt. Es ist nicht möglich, hier einen Dateinamen als Argument zu übergeben. **leavemk** vergißt die geladene Make-Datei und schaltet wieder in den Normal-Modus.

Zur Manipulation von Projekten stehen nur die beiden Funktionen zur Verfügung, die sich sinnvoll von einem Editor aus starten lassen: **"addtx <Quelltextname>"** fügt eine neue Quelltextdatei hinzu, und **"addtxobj <Quelltextname>"** trägt eine Quelltextdatei samt passender Objektdatei und geeigneter Compileranweiung in die Make-Datei ein. Dabei werden natürlich auch auf *„.asm"* endende Quelltexte sinnvoll behandelt.

Alle anderen Operationen kann der Anwender über das Make-Dialogfenster vornehmen. Es wird mit **openmake** geöffnet.

2.6.3.5 Schreiben von Dateien

Mit **"exefile <Dateiname>"** wird - falls ein übersetztes und gelinktes Programm vorliegt - eine ausführbare Programmdatei geschrieben, während **"objectfile <Dateiname>"** eine linkbare Objektdatei schreibt. Bei beiden Anweisungen ist der Dateiname optional. Wird er weggelassen, öffnet sich ein Dateiauswahldialog.

2.6.3.6 Optionen speichern und setzen

Die beiden Anweisungen **"saveconf <Dateiname>"** **"savesession <Dateiname>"** schreiben alle Einstellungen in die angegebene Datei. Der einzige Unterschied ist der Default-Name, den die Anweisungen verwenden, wenn kein Dateiname angegeben wird - dies sind nämlich *„s:mcpp.config"* und *„mcpp.SESSION"*. Systemeinstellungen können aus einem ARexx-Script aber auch beliebig gesetzt werden. Dazu dient die Anweisung **"setopt <Optionen>"** Als Argument ist eine beliebig lange Folge von Einstellungen erlaubt. Diese sind in dem Format anzugeben, in dem sie in der Konfig- und in den Session-Dateien stehen.

Beispiel:

```
"setopt -X21,"hello World!" -pp -b -X10,1"
```

setzt den Argumentstring für Programmstarts auf `"hello World!"` und schaltet als Default in den C-Modus, aktivert die Erzeugung von Debug-Dateien und läßt bei jeder Übersetzung automatisch eine Exe-Datei schreiben.

Die Compiler- und Linker-Einstellungen wie `"-pp"` oder `"-b"` sind identisch mit den Optionen des CLI-Compilers „mcppc". Alle Optionen, die die Entwicklungsumgebung betreffen, werden mit einer mit `"-x"` beginnenden Anweisung eingestellt und werden in Abschnitt 2.5.5 dieses Handbuchs erläutert.

2.6.3.7 Informationen über die Entwicklungsumgebung

Für elegante Lösungen sollten Scripts auch in der Lage sein, Informationen über den aktuellen Status von *MCPP* zu erhalten. `"getinfo MAKE"` liefert den Namen der momentan geladenen Make-Datei bzw. einen Leerstring. Mit `"getinfo STATE"` erhalten Sie einen String **„NONE"**, **„COMPILED"** oder **„LINKED"** und können so feststellen, ob ein übersetztes bzw. übersetztes und gelinktes Programm vorliegt. In beiden Fällen liegt er Ergebnisstring nachher in der Variablen `"RESULT"`. Es ist auch zu beachten, daß vorher mit der Rexx-Anweisung `Options results` die Datenübergabe von *MCPP* in das ARexx-Programm ermöglicht wurde.

2.6.3.8 Fehlermeldungen und Fehlerstellenansprung

Die Anweisung `"errorscript <dateiname>"` legt ein ARexx-Script fest, das von *MCPP* jedesmal aufgerufen wird, wenn der Compiler eine Fehlermeldung ausgibt. Diesem Script werden als Argumente Dateiname, Zeilen- und Spaltennummer sowie die eigentliche Fehlermeldung übergeben. So veranlaßt `"errorscript errors.rx"` *MCPP* bei jedem folgenden Compilerfehler zu einem Aufruf der Art:

```
rx errors.rx "hello.c" 8 1 "Syntax Error"
```

Ihr Script `"errors.rx"` kann dann mit den übergebenen Daten beliebiges tun, z. B. sie in einer Fehlerdatei abspeichern.

Ein `"errorscript"`-Aufruf ohne Argument schaltet diesen Aufrufmechanismus wieder ab.

Da die Fehlerdaten schlicht an das Argument von `"errorscript"` angehängt werden, können Sie trickreich auch weitere Argumente an das ARexx-Script übergeben:

```
"errorscript wagga.rx ERRORS argl! bläk"
```

bewirkt Aufrufe wie:

```
rx wagga.rx ERRORS argl! bläk "hello.c" ...
```

Analog funktioniert die Anweisung `"editscript <Scriptname>"`, die dafür gedacht ist, auch im Rexx-Modus Fehlerstellenansprung zu realisieren. Das angegebene Script wird aufgerufen, wenn der Benutzer im Compiler-Fenster nach Ansprung einer Fehlerposition verlangt, und erhält als Argumente Dateinamen, Zeilen- und Spaltennummer, z. B.:

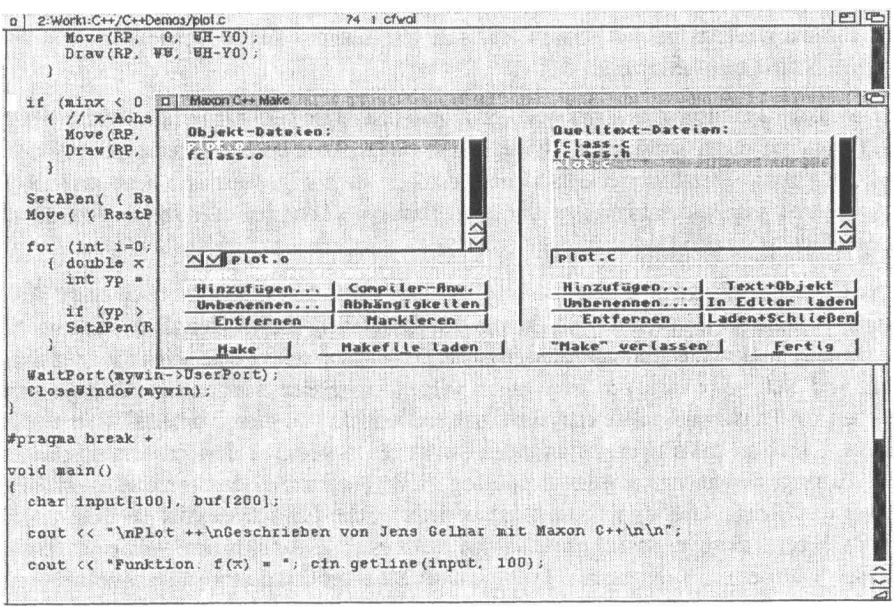
```
rx edit.rx "hello.c" 8 1
```

3. MAKE - eine integrierte Projektverwaltung

3.1 Das Prinzip

3.1.1 Wozu eine Projektverwaltung?

Größere Projekte bestehen normalerweise nicht aus einer einzelnen Quelltextdatei, sondern aus einer Vielzahl von Modulen, die vom Linker verbunden werden - zumindestens sollte es so sein. Das nimmt leicht Ausmaße an, die nicht mehr ganz leicht zu überschauen sind. Hinzu kommt, daß es natürlich Unsinn ist, nach jeder Änderung das ganze Projekt zu übersetzen. Auf die Dauer wird es aber lästig, sich immer wieder neu zu überlegen, welches Modul jeweils übersetzt werden muß, einmal ganz davon abgesehen, daß man eigenartige Überraschungen erleben kann, wenn man dabei etwas vergißt. So entsteht das Bedürfnis, solche Vorgänge zu automatisieren - wozu hat man schließlich einen Computer...



Die populärste Projektverwaltung, die auf so ziemlich allen ernstzunehmenden Rechnern anzutreffen und bei allen Compilersprachen tausendfach bewährt ist, heißt „make". Eigentlich ist „make" selbst eine Art Programmiersprache, und das macht es natürlich nicht leicht, ein komplettes Make in eine graphisch orientierte Entwicklungsumgebung wie die von MaxonC^{++} zu integrieren. Deshalb unterstützt *MCPP* nur einen recht begrenzten (aber den entscheidenden) Teil von „make". Übrigens wurde das oft kritisierte „make" von MaxonC^{++} 1.x vollständig überarbeitet und sollte jetzt kaum noch Wünsche offen lassen.

3.1.2 Was ist das überhaupt - ein Projekt?

Fangen wir doch einmal ganz graswurzelmäßig von vorn an: Woraus besteht ein Projekt (immerhin war von einer Projekt-Verwaltung die Rede) - und was ist überhaupt ein Projekt?

Also, zu den oben erwähnten Einschränkungen gehört, daß ein Projekt immer genau ein ausführbares Programm darstellt. Wenn man im Rahmen eines Projekts ein Programm in mehreren Versionen erstellen will (man denke da nur an „MCPP" und „mcppc", die im wesentlichen ein und das selbe Programm sind, nur mit unterschiedlichen aufgepfropften Benutzerschnittstellen), muß man eben für jedes einzelne Executable eine eigene Projektdatei erstellen.

OK - ein Projekt ist also ein ausführbares Programm. Und wie kommt man zu einem solchen Programm, sofern es sich nicht gerade um ein Bonsai-Miniatur-Projekt handelt?

Richtg, das erwähnte ich schon: Man linkt dafür Objektdateien, also Module, zusammen. Genaugenommen bindet der Linker auch noch fertige Bibliotheksmodule ein, aber das interessiert uns hier nur höchst periphär, oder genaugenommen nicht die Bohne. Der Freizeit-Existenzialist stellt sich jedenfalls nun die Frage „Und wo kommen die kleinen Module her?", der Coder weiß es schon: Das hat ausnahmsweise nichts mit Schweinskrams zu tun, sondern die Objektdateien werden von irgendwelchen Compilern erzeugt.

Zu jedem der Module gibt es genau einen ganz besonderen Quelltext, nämlich den Text, aus dem die Objektdatei durch Compilieren entsteht. Um so ein Objekt zu machen, braucht die Projektverwaltung also im wesentlichen eine Information darüber, wie der Quelltext dazu heißt, und außerdem noch ein paar Zusatzinformationen, etwa den Namen des Compilers oder irgendwelche der allseits beliebten Compiler-Optionen.

Aber das ist noch lange nicht alles, denn zu einemnormalen Projekt gehören noch einige Texte mehr - normalerweise Includedateien, die von den eigentlichen Quelltexten irgendwo und irgendwie benutzt werden. Das führt uns natürlich zu der Frage, wann und wie eine Objektdatei neu übersetzt wird. Nun, bevor wir alle in Rente gehen, werden wir vielleicht noch das Entwicklungssystem erleben, das Programme on-line compiliert, während man den Quelltext bearbeitet. Vorerst ist es jedenfalls noch so, daß man munter an seinen Quelltexten 'rumediert und irgendwann entscheidet, daß das Programm für eine erneute Übersetzung reif ist. Dann tritt die bereits mehrfach erwähnte Projektverwaltung „Make" in Aktion und schaut nach, welche Objektdateien für das Projekt nötig sind, und bevor diese zusammengelinkt werden, denkt es scharf darüber nach, welche der Objektdateien neu übersetzt („recompiliert") werden muß. Da ja der Compiler wenigstens zeitweise mit dem Programmierer Schritt halten soll, ist auch noch halbwegs einsichtig, wann das zu geschehen hat: Richtig, genau dann, wenn mindestens einer der Texte, aus denen das Modul entsteht (d. h. Quelltext und Include-Files), seit der letzten Übersetzung verändert wurde. Das läßt sich leicht an Erstellungsdatum und -uhrzeit der entsprechenden Dateien feststellen.

3.1.3 Was hat das ganze mit „Make" zu tun?

Die Projektverwaltung verwaltet im wesentlichen zweierlei Datenbestände: Texte und Objekte. Bei den Texten handelt es sich im allgemeinen um irgendwelche Quelltexte oder Includedateien, und aus diesen Texten erzeugt der Compiler unter der Kontrolle der Projektverwaltung die Objektdateien. Im letzten Schritt werden diese Objektdateien dann mit den Libraries zu einer ausführbaren Datei gelinkt.

Für jede der Objektdateien braucht das Make also Angaben darüber, wie es „gemacht" (daher der Name) werden soll, insbesondere den Namen eines Quelltexts und eines Compilers. Das würde man natürlich auch noch von Hand hinkriegen, und deshalb sagt man der Projektverwaltung auch, unter welchen Umständen ein Objekt compiliert werden soll und wann die alte Version noch einmal verwendet werden kann.

Wie bereits gesagt, muß ein Objekt neu übersetzt werden, wenn ein Text, der mit diesem Objekt zu tun hat, verändert wurde. Normalerweise handelt es sich bei diesen Texten um den Quelltext des Moduls sowie die darin benutzten Includedateien. Dazu vergleicht Make den Datestamp, also die Zeit der letzten Veränderung einer Datei, der Objektdatei mit denen „ihrer" Texte. Natürlich wird ein Objekt auch dann compiliert, wenn es bisher noch nicht existierte.

3.2 Die Praxis

3.2.1 Erstellen von Projekten

Alle Angaben über die Struktur eines Projekts werden sinnvollerweise in einer Datei festgehalten, dem Makefile. Mit MaxonC^{++} erstellen Sie diese Datei interaktiv mit dem Pull-Down-Menü **„Make"** und dem Make-Dialogfenster. Das Makefile wird dabei ständig aktualisiert, also nach jeder Veränderung neu abgespeichert. Da eine solche Datei normalerweise nicht sehr lang sein wird, dürfte dabei auch beim Diskettenbetrieb keine Verzögerung auftreten.

Am besten probieren Sie das Make-Utility einfach aus, indem Sie sich ein Mini-Projekt aus zwei Modülchen erstellen. Damit fangen Sie an, indem Sie den Menüpunkt **„Neues Makefile"** anwählen. Wenn bisher keine Projektdatei geladen ist, können Sie ebenso gut auch den Menüpunkt **„Projektverwaltung"** benutzen. Nun erscheint der Dateiauswahl-Dialog und fragt Sie nach dem gewünschten Dateinamen. Meist nennt man die Datei ganz einfach *„makefile"*, jedenfalls dann, wenn man nicht mehr als eine solche Datei in einem Verzeichnis hat. Wenn Sie dagegen an einem vorhandendem Projekt weiterarbeiten wollen, benutzen Sie den Menüpunkt **„Makefile laden"**.

Nach dem Erzeugen einer neuen Projektdatei öffnet sich ein Fenster, das zwei Scrollisten (eine mit ***„Objekt-Dateien"***, die andere mit ***„Quelltext-Dateien"*** betitelt) und eine ganze Menge Gadgets enthält. Klicken Sie hier am besten einfach auf das Gadget ***„Fertig"*** in der rechten unteren Ecke, um das Fenster wieder zu schlieflen.

An einem Projekt, daß mehr als einen Quelltext umfaßt, wird man in der Regel länger als einen Tag lang arbeiten. Deshalb sollten Sie die folgenden Tips beachten, um es dabei auf lange Sicht komfortabel zu haben:

- ☞ Zu jedem Projekt gehört ein eigenes Verzeichnis. Sie sollten also auf jeden Fall mit der Workbench oder Shell ein neues Unterverzeichnis in Ihrem *„MCPP:"*-Verzeichnis oder auch anderswo anlegen.
- ☞ Wenn Sie *MCPP* immer von der Shell starten, sollten Sie vorher stets in das Verzeichnis wechseln, in dem das *Makefile* liegt. Dadurch wird dies auch für MCPP und *Edward* zum aktuellen Verzeichnis, und dadurch werden die *SESSION-Dateien* von dort gelesen bzw. dahin geschrieben. Insbesondere werden dann auch die Compiler-Einstellungen von *MCPP* im passenden Verzeichnis abgelegt und beim nächsten Start von *MCPP* automatisch wieder eingeladen.
- ☞ *MCPP* kann von der Workbench durch Anklicken des Makefile-Icons gestartet werden. Die Projektdatei wird dabei automatisch als solche erkannt und geladen (und nicht etwa als Textdatei in den Editor). Dabei wird das Verzeichnis, in dem das Makefile liegt, zum aktuellen Verzeichnis von *MCPP* und *Edward*, was wiederum erfreuliche Auswirkungen auf die Behandlung der *SESSION-Dateien* hat (siehe oben).

Die bequemste Art, sich ein Projekt für MaxonC++ einzurichten, geht deshalb so:

1. Stellen Sie sicher, daß die Option ***„Icons erzeugen"*** eingeschaltet ist und daß ein geeignetes leeres Verzeichnis für Ihr Projekt existiert.
2. Erzeugen Sie in diesem Verzeichnis wie oben beschrieben eine Make-Datei, die Sie zunächst einmal so leer lassen, wie sie ist.
3. Beenden Sie jetzt *MCPP*.
4. Wenn Sie Workbench 2.0 (oder später) benutzen, machen Sie jetzt am besten ein **„Leave Out"** (**auslagern**) des Makefile-Icons. Dadurch liegt es immer sichtbar auf dem Workbench-Bildschirm und kann nach dem Einschalten des Rechners sofort geöffnet werden.
5. Starten Sie nun *MCPP*, indem Sie das Makefile-Icon (natürlich doppelt) anklicken. Nun werden *„Edward.SESSION"* und *„mcpp.SESSION"* automatisch aus dem Verzeichnis des Projekts gelesen (bzw. dorthin geschrieben).

Vielleicht erscheint Ihnen diese Prozedur etwas umständlich, aber auf die Dauer werden Sie es zu schätzen wissen, nach dem Booten des Rechners mit einem einfachen Doppelklick genau da weiterarbeiten zu können, wo Sie zuletzt aufgehört haben.

3.2.2 Hinzufügen von Dateien, umständliche Methode

Zu einem Projekt gehört natürlich mindestens eine Objektdatei, und um eine solche zu erzeugen, braucht man einen Quelltext. Versuchen wir es zunächst auf die umständliche Methode und schreiben einen Text, z. B.

```
#include <stream.h>
void test()  { cout << "Dies ist ein Test."; }
```

Diesen Text speichern Sie einmal als „*test.c*" ab (mit einer Datei ohne Namen kann Make nichts anfangen) und wählen dann den Menüpunkt **„Make/Projektverwaltung"**. Damit öffnen Sie wieder das Make-Fenster, das Sie schon beim Erzeugen der Projektdatei gesehen haben. Dabei ist Ihnen vielleicht auch schon aufgefallen, daß das Fenster vertikal zweigeteilt ist, wobei die linke Hälfte mit ***„Objekt-Dateien"*** und die Rechte mit ***„Quelltext-Dateien"*** überschrieben ist.Wenn Sie nun die Textdatei „*test.c*" und die Objektdatei „*test.o*" in Ihr Projekt aufnehmen möchten, geschieht dies eigentlich ganz intuitiv: Klicken Sie zunächst auf das Gadget ***„Hinzufügen..."*** in der rechten Spalte. Es öffnet sich nun ein Dateiauswahlfenster, in dem Sie die soeben erzeugte Datei „*test.c*" abwählen. Sie bestätigen die Auswahl, der Datei-Dialog wird geschlossen und schon steht „*test.c*" in der rechten Liste des Make-Fensters. Ebenso einfach wählen Sie jetzt das linke Gadget ***„Hinzufügen..."*** an und wählen in der Dateiauswahl den Dateinamen „*test.o*". Höchstwahrscheinlich existiert diese Datei noch überhaupt nicht, so daß Sie sie folglich nicht einfach anklicken können, sondern den Namen von Hand eintippen müssen.

Natürlich spielt es keine Rolle, ob sie zuerst - wie hier beschrieben - die Quelltextdatei und dann die Objektdatei oder umgekehrt eintragen. Die beiden Listen sind zunächst vollkommen voneinander unabhängig, und wenn Sie möchten, können Sie jetzt auf beiden Seiten beliebig viele Dateien eintragen, die allesamt auf der Platte vorhanden sein können, aber nicht müssen. Die Dateien müssen auch keineswegs im selben Verzeichnis wie die Make-Datei liegen, aber das ist allgemein üblich und wohl auch am praktischsten.

Wenn Sie jeweils mehrere Dateien aufnehmen, wird Ihnen auffallen, daß die Namen der Quelltext-Dateien alphabetisch sortiert werden, während die Objektdateien in der Reihenfolge erscheinen, in der Sie hinzugefügt werden. Dies ist auch die Reihenfolge, in der der Linker später die Objektdateien aneinanderhängt, was z. B. dann von Interesse ist, wenn man einen eigenen Startup-Code schreiben möchte. Die Reihenfolge der Objektdateien können Sie mit den beiden Pfeilgadgets links von der Anzeige für die angewählte Objektdatei verändern. Dumm wie Computer nun einmal sind, müssen Sie nun aber auch noch klarmachen, daß die Objektdatei „*test.o*" entsteht, indem „*test.c*" compiliert wird. Wählen Sie dazu „*test.o*" in der linken Spalte an (so daß der Name invertiert dargestellt wird und in der Anzeige unter der Objekt-Liste erscheint) und klicken Sie dann auf das Gadget ***„Compiler-Anw."***. Alternativ können Sie „*test.o*" auch mit einem Doppelklick anwählen. Nun öffnet sich ein neues Fenster, in dem alle Gadgets bis auf ein Cycle-Gadget oben links und die üblichen ***„OK"***- und ***„Abbruch"***-Gadgets am unteren Rand deaktivert sind. Das Cycle-Gagdet zeigt zunächst ***„Nicht übersetzbar"*** an. Nach einem Klick erscheint hier ***„intern compilieren"***, und alle Gadgets bis auf ein Texteingabefeld unten links werden gleichzeitig aktiv. Insbesondere kann jetzt auch die mit ***„Quelldatei"*** überschriebene Liste benutzt werden. Dort wählen Sie jetzt einfach „*test.c*" als Quelltext an und schließen das Fenster wieder mit einem Klick auf ***„OK"***.

Im Make-Fenster erscheint jetzt direkt unterhalb der Objekt-Liste „*mcppc -c test.c*". Dies ist nichts anderes als die Anweisung, mit der man in der Shell „*test.c*" nach „*test.o*" übersetzen würde.

3.2.3 Übersetzen des Projekts

Sie können das Programm jetzt compilieren, indem Sie entweder im Make-Fenster auf das Gadget *„Make"* klicken oder wie gewohnt im Editor die Taste <F9> bzw. das Menü **„Compiler/Übersetzen"** anwählen. Nun tritt Make in Aktion, stellt fest, daß *„test.o"* compiliert werden muß, und bindet anschließend aus diesem Objectfile ein Programm zusammen - oder versucht es zumindest, denn es gibt ja noch keine „main"-Funktion. Make funktioniert völlig unabhängig davon, welcher Text sich gerade im Editor befindet.

3.2.4 Hinzufügen von Dateien, einfache Methode

Beim folgenden Miniatur-Modul, das die noch fehlende „main"-Funktion enthält, machen wir es uns etwas einfacher:

```
#include "test.h"
void main()  {    test();  }
```

Diesen Text speichern wir einfach als *„main.c"* ab und wählen den Menüpunkt **„Text und Objekt hinzufügen"**. Jetzt macht Make vollautomatisch folgendes:

- ☞ Die Datei *„main.c"* wird in die Liste der Texte aufgenommen.
- ☞ Das Objekt *„main.o"* wird dem Projekt hinzugefügt.
- ☞ Die Compileranweisung für *„main.o"* wird passend gesetzt.

Jetzt fehlt dem Projekt nur noch eins: Die Header-Datei *„test.h"*, in der die Funktionen (na ja, hier ist es ja nur eine) des Moduls *„test"* deklariert. Die Datei enthält lediglich folgendes:

```
void test();
```

also den Prototypen der Funktion. So, wenn Sie jetzt den Compiler starten, müßte das Projekt korrekt übersetzt und gelinkt werden.

Zuvor sollten Sie aber *„test.h"* noch als Quelltext dem Projekt hinzufügen (im folgenden Abschnitt verrate ich Ihnen auch, wozu das gut sein soll). Das können Sie wie gehabt vom Make-Fenster aus erledigen. Wenn Sie die Datei aber sowieso gerade im Editor haben, können Sie auch den Menüpunkt **„Make/Text aus Editor aufnehmen"** benutzen, und schon wird der aktuelle Text des Editors in die Quelltext-Liste aufgenommen.

3.2.5 Abhängigkeiten

Das so ziemlich wichtigste Feature der Projektverwaltung haben wir aber bisher nicht benutzt: Die Abhängigkeiten. Offensichtlich benutzt in unserem Beispiel *„main.o"* die Datei *„test.h"* als Includedatei und ist folglich davon abhängig. Immer, wenn *„test.h"* verändert wurde, sollte deshalb auch *„main.o"* sicherheitshalber neu übersetzt werden.

Öffnen Sie also wieder das Make-Fenster, wählen Sie in der linken Liste die Objektdatei *„main.o"* an und klicken Sie dann auf das Gadget ***„Abhängigkeiten"***. Dabei werden Sie bemerken, daß dieses Gadget invertiert bleibt, bis Sie es erneut anklicken. Nun können Sie in der Quelltext-Liste alle die Dateien anwählen, von denen *„main.o"* abhängig sein soll, also insbesondere *„test.h"*. Korrekterweise können Sie auch *„main.c"* anwählen, aber bei interner Übersetzung weiß MaxonC^{++} von selbst, daß jede Objektdatei von ihrer Quelltextdatei anhängt. Jedenfalls werden Sie dabei merken, daß Sie jetzt in der rechten Spalte beliebig viele Zeilen gleichzeitig aktivieren und durch erneutes Anklicken wieder deaktiveren können. Dieser Modus bleibt erhalten, so lange das Gadget ***„Abhängigkeiten"*** invertiert ist. Sie können jetzt auch ganz einfach auf eine andere Objektdatei klicken, um deren Abhängigkeiten zu betrachten und zu manipulieren.

Im Grunde genommen ist dieses manuelle Setzen von Anhängigkeiten aber überhaupt nicht nötig. Es reicht völlig, wenn Sie im System-Dialogfenster (Menüpunkt **„Optionen/System"**) das vielversprechend mit ***„Make-Datei: Abhängigkeiten automatisch setzen"*** beschriftete Symbol anwählen. Wann immer nun eine Objektdatei des Projekts compiliert wird, merkt *MCPP* sich, welche Includedateien dabei gelesen wurden - natürlich nur die selbstgeschriebenen, die man mit `#include "Name"` einbindet, und nicht etwa die System-Includes, die mit `#include <Name>` includiert werden, denn jene haben in einer Make-Datei normalerweise nichts zu suchen. Anschließend werden die Abhängigkeiten der jeweiligen Objektdatei automatisch gesetzt, wobei nötigenfalls sogar neue Textdateien in das Projekt aufgenommen werden.

Deshalb beschränkt sich das Anlegen eines Projekts zusammengefaßt auf folgende Schritte:

- ☞ Erzeugen sie eine neue Projektdatei.
- ☞ Laden Sie alle „echten" Quelltextdateien, also keine Includefiles, einmal in den Editor und wählen Sie dann den Menüpunkt **„Text und Objekt aufnehmen"**.
- ☞ Starten Sie die Übersetzung, wobei Sie sicherstellen sollten, daß die Objektdateien allesamt noch nicht existieren. Dann werden selbsttätig alle Quelltexte durchcompiliert, die Includedateien in das Projekt aufgenommen und die Abhängigkeiten korrekt gesetzt.

3.2.6 Weitere Funktionen im Make-Fenster

3.2.6.1 Projektpflege

Es kommt hin und wieder vor, daß eine Quelltext- oder Objektdatei komplett überflüssig wird. Dann wählen Sie die betreffende Datei einfach auf der richtigen Seite des Make-Fensters an und klicken auf das Gadget ***„Entfernen"***. Nach einer Sicherheitsabfrage wird die Datei vollständig aus dem Projekt entfernt.

Ebenso kann es passieren, daß Ihnen ein Dateiname plötzlich nicht mehr ganz passend erscheint. Dies können Sie elegant erledigen, indem Sie den Dateinamen im Make-Fenster anklicken und dann das Gadget ***„Umbenennen..."*** benutzen (aber bitte das auf der richtigen Seite, denn auch diese Funktion gibt es gleichermaßen für Objekt- und Quelldateien). Es erscheint der übliche Datei-Dialog, dem Sie nun den gewünschten neuen Namen anvertrauen. Beachten Sie aber bitte, daß diese

Funktion die Datei ausschließlich in der Make-Datei umbenennt. Das eigentliche Umbenennen der Datei auf der Festplatte bzw. Diskette müssen Sie schon selbst erledigen.

3.2.6.2 Übersetzung erzwingen

Eigentlich entscheidet das Make ja selbständig und prinzipiell auch korrekt, welche Datei wann recompiliert werden soll. Manchmal möchte man dies aber auch erzwingen, z. B. wenn man Compileroptionen verändert hat und einige oder alle Objektdateien gezielt mit diesen neuen Optionen übersetzen lassen möchte. Deshalb gibt es im Make-Fenster das Gadget ***„Markieren"***.

Ähnlich wie ***„Abhängigkeiten"*** ist auch ***„Markieren"*** ein Umschaltbares Gadget. Während es aktiviert ist, können Sie in der Objektliste beliebig Dateien anwählen und deaktivieren. Beim nächsten Übersetzungsvorgang werden die so markierten Dateien dann auf jeden Fall compiliert, und zwar zusätzlich zu den Dateien, die Make ohnehin für übersetzungswürdig befindet.

3.2.6.3 Dateien in den Editor laden

Beim Programmieren hat man es fast immer nur mit den Textdateien zu tun, die zum aktuellen Projekt gehören. Deshalb bietet *MCPP* eine elegante Möglichkeit, Quelltexte in den Editor zu laden.

Öffnen Sie dazu das Make-Fenster und wählen Sie in der rechten Liste den gewünschten Quelltext an. Wenn Sie nun das Gadget ***„In Editor laden"*** anklicken, wird der Text - wer hätte das gedacht - in den *Edward* hinzugeladen. Das Gadget ***„Laden+Schließen"*** funktioniert genauso, schließt aber gleichzeitig das Make-Fenster. Dies könenn Sie auch mit einem Doppelklick auf den Dateinamen erreichen.

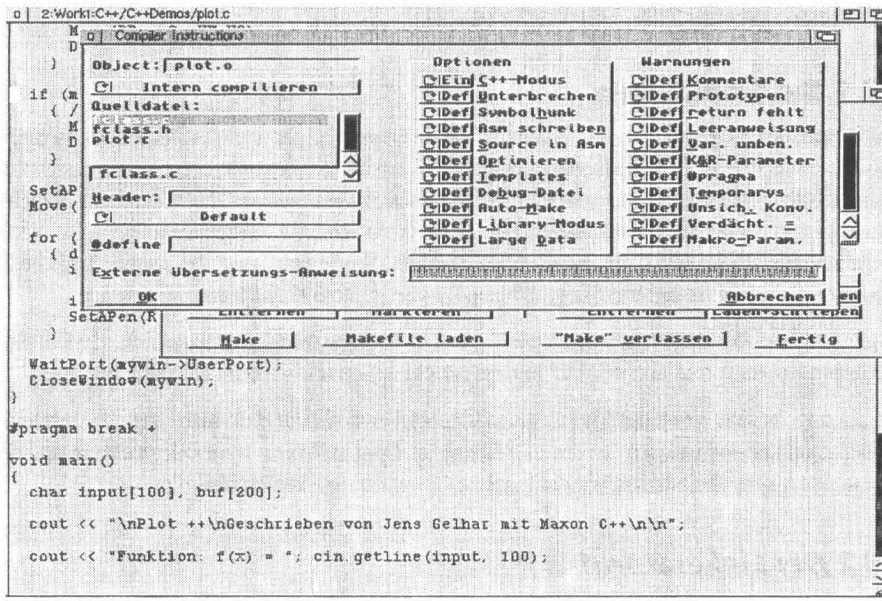
3.2.6.4 Projektdatei wechseln

Am unteren Rand des Make-Fensters befinden sich zwei bisher unerwähnte Gadgets: ***„Makefile laden"*** und ***„Make verlassen"***. Naheliegenderweise erscheint bei Anwahl des ersteren die Datei-Auswahlbox und ermöglicht Ihnen, eine andere Projektdatei zu laden. Mit dem anderen Gadget verlassen Sie den Make-Modus und gelangen wieder in die normale Betriebsart für Ein-Modul-Programme.

3.2.7 Compileranweisungen für Objektdateien

3.2.7.1 Interne Übersetzung

Zwischen dem Compileranweisungs-Requester der Projektverwaltung und dem globalen Compiler-Dialogfenster besteht auch optisch eine gewisse Ähnlichkeit. Betrachen Sie einmal die Gadgets, die benutzbar werden, wenn für eine Objektdatei der Modus ***„Intern compilieren"*** angewählt wurde. Sie werden feststellen, daß die wichtigsten Compileroptionen für jedes Modul des Projekts separat eingestellt werden können.



Allgemein gelten die globalen Einstellungen auch im Make-Modus, sofern eine Option nicht explizit anders eingestellt wird. Deshalb finden Sie im Compileranweisungs-Requester eine ganze Menge kleiner Cycle-Gadgets mit den Stellungen *„Def"*, *„Ein"* und *„Aus"*. Dabei steht *„Def"*, wie man sich denken kann, für *„Default"*, d. h. bei dieser Schalterstellung soll für das Modul die globale Einstellung gültig sein.

Bei den mit *„Optionen"* überschriebenen Gadgets entsprechen die ersten acht exakt den entsprechenden globalen Schaltern. *„Auto-Make"* entspricht einem Gadget aus dem „System"-Dialog und legt fest, ob die Abhängigkeiten der betreffenden Objektdatei vom Make automatisch gesetzt werden sollen, *„Library-Modus"* übersetzt ein Modul in handlichen kleinen Häppchen (siehe Einstellung *„Library erzeugen"* in Abschnitt 2.5.3.2), und *„Large Data"* entspricht der Einstellung *„Datenmodell: Groß"* im globalen Linker-Requester.

Die elf mit *„Warnungen"* überschriebenen Einsteller entsprechen natürlich exakt den elf globalen optionalen Warnings (siehe 2.5.2.5), und im *„#define"* überschriebenen Texteingabefeld kann optional ein zusätzliches Makro definiert werden. Es ist also nicht möglich, ein global definiertes Symbol für ein einzelnes Modul umzudefinieren.

Natürlich fehlen noch ein paar Compileroptionen, z. B. ist es über die Dialoge nicht möglich, für eine Übersetzungseinheit die Include-Pfade anders zu setzen. Solche (wohl eher ungewöhnliche) Einstellungen sind aber möglich, wenn Sie die Projektdatei in den Editor laden, „von Hand" verändern, abspeichern und dann erneut als Makedatei laden. In Abschnitt 3.3 erfahren Sie, wie die Projektdateien aufgebaut sind.

3.3 Handgemachte Makedateien

3.3.1 Die Grundlagen

Das Format der Projektdateien von MaxonC++ orientiert sich an den Standard-Makefiles, ist aber natürlich etwas strikter, da in der integrierten Entwicklungsumgebung ja nur ein gewisser Ausschnitt des bekannten Make implementiert werden konnte. Es ist dadurch relativ leicht, ein solches Makefile von Hand zu erstellen oder ein vorhandenes Makefile an MaxonC++ anzupassen. Wenn Sie nicht in der integrierten MCPP-Umgebung arbeiten, sondern die Option `"-m"` des mcppc benutzen, bleibt Ihnen sogar nichts anderes übrig, als Make-Dateien grundsätzlich manuell zu erstellen.

Es gibt zwei verschiedene Arten von Zeilen, nämlich Abhängigkeits- und Anweisungszzeilen. Sie unterschieden sich formal dadurch, daß letztere mit einem Leerzeichen oder Tabulator beginnen.

Ein „Eintrag" beginnt grundsätzlich mit einer Abhängigkeitszeile, die dem Make sagt, was erzeugt werden kann und von welchen Dateien es abhängig ist. Optional können eine oder mehrere Anweisungszeilen folgen, die festlegen, wie das fragliche Objekt erzeugt werden muß.

3.3.2 Der Linkeraufruf

MaxonC++ erwartet als ersten Eintrag stets die Exe-Datei, um die es geht (Sie erinnern sich: Leider immer nur ein Exefile pro Makefile). Die Exe-Datei ist - formal gesehen - von den Objektdateien abhängig, und erzeugt wird sie durch einen Linker-Aufruf. Als Linker muß hier der im *mcppc* benutzt werden (auch wenn Sie nicht *mcppc*, sondern *MCPP* verwenden). In diesem Zusammenhang sollte ich wohl erwähnen, daß es für das Verständnis dieses Kapitels enorm hilfreich wäre, wenn Sie schon ein wenig mit dem *mcppc* vertraut wären (siehe sonst Kapitel 5).

Eine Abhängigkeitszeile beginnt stets mit dem Namen der zu erzeugenden Datei, gefolgt von einem „:" und optional der Liste der Abhängigkeiten. Wenn unser Projekt „*Demo*" heißt und aus den beiden Modulen „*main.o*" und „*test.o*" besteht, beginnt unser Makefile also mit den Zeilen:

```
Demo: main.o test.o
 mcppc main.o test.o -o Demo
```

Dazu noch einige Anmerkungen:

☞ Aus Gründen der Kompatiblität zur ersten Version ist statt `"mcppc"` auch `"cppc"` erlaubt. Dieses Feature wird aber möglicherweise in späteren Versionen von MaxonC++ nicht mehr enthalten sein.

☞ Da das MaxonC++ Make keinen externen Linker aufruft, sondern alles implizit intern erledigt, ist auch die Option `"-o Demo"`, die den Namen der Exe-Datei festlegt, strenggenommen nicht erforderlich, zumal er aus der Abhängigkeitszeile bereits bekannt ist. Aus Ästhetischen Gründen sollte man diese Angabe aber trotzdem machen.

☞ MaxonC++-Make weiß, daß ein Linkeraufruf von den benutzten Objektdateien abhängig ist. Deshalb sind auch die beiden Einträge in der Abhängigkeitszeile nicht wirklich nötig.

3.3.3 Internes Compilieren

Nach dem Linkeraufruf folgen in beliebiger Reihenfolge die Einträge für die einzelnen Objektdateien. Wenn Ihnen danach ist, können Sie auch Einträge für Projektdateien hinzufügen, die überhaupt nicht zum Projekt gehören - die werden dann ignoriert.

Auch hier beginnt jeder Eintrag mit einer Abhängigkeitszeile, also z. B.

```
main.o: main.c test.h
```

Die Objektdatei *„main.o"* soll also bitteschön übersetzt werden, wenn eine der Dateien *„main.c"* und *„test.h"* in letzter Zeit verändert wurde. Fehlt noch der Compileraufruf: Der Compiler heißt ja bekanntlich genau wie der Linker *„mcppc"* (oder auch *„cppc"*, siehe oben). Die Option `"-c"` sorgt dafür, daß eine Objektdatei statt einer Exe-Datei erzeugt wird:

```
main.o: main.c test.h
  mcppc -c main.c
```

Außer `"-c"` kann auch jede andere Compiler-Option aus Kapitel 5 benutzt werden, z. B. `"-O"` zum Einschalten der Optimierungen. Bei allen Optionen, die nicht explizit gesetzt werden, werden die globalen Defaults verwendet.

Als Erbstück aus der 1.x-Version versteht MaxonC++ Make hier auch zwei alternative Compiler-Anweisungen:

`"cc"` ist gleichbedeutend mit `"mcppc -c -pc"` und

`"c++"` entspricht `"mcppc -c -pp"`. Dabei wird also explizit im ANSI C- bzw. im C++-Modus compiliert.

Es ist übrigens nicht zwingend erforderlich, dem Make zu sagen, daß ein Objekt von seiner Quelltextdatei (und das gesamte Projekt von seinen Objektdateien) abhängig ist, denn das weiß Make auch so. Also könnte man das Makefile folgendermaßen abkürzen:

```
Demo:  mcppc main.o test.o
test.o:
  mcppc -c test.c
main.o: test.h
  mcppc -c main.c
```

3.3.4 Externe Übersetzung

Jede Compileranweisung, die mit *„mcppc"* beginnt, wird von MaxonC++ „intern" ausgeführt, d. h. Make und Compiler sind Bestandteile eines einzigen Programms und keines von beiden muß

irgendwie nachgeladen werden. Leider ist MaxonC++ zugegebenermaßen nicht ausreichend (z. B. hat man öfters ein Assembler-Modul in seinem Projekt). Aber auch das ist kein Problem: Schreiben Sie die Anweisung ganz einfach in das Makefile! Sie können dann sogar mehrere nacheinender auszuführende Zeilen (=Shell-Anweisungen) für eine Objektdatei angeben, z. B. so:

```
sub.o: sub.asm
  mcpp:bin/masm sub.asm -c clos    echo "Assembler ist fertig"
```

Wenn Sie diese Projektdatei von der integrierten Umgebung MCPP ausführen lassen, wird vorher sogar geprüft, ob die Dateien, von denen das Objekt abhängig ist (in diesem Falle nur „*sub.asm*"), sich im Editor befinden und seit ihrer letzten Abspeicherung verändert wurden. Falls ja, werden sie automatisch abgespeichert, denn der externe Compiler/Assembler liest seinen Quelltext ja aus der Datei und nicht (wie MaxonC++) direkt aus dem *Edward*.

Speziell bei der Benutzung des Maxon Assemblers gibt es noch eine Besonderheit: Nach der Übersetzung schaut *MCPP* nach, ob eine zur Objektdatei passende „*.err*"-Datei erzeugt wurde (im Beispiel also „*sub.err*"). Falls ja, versucht *MCPP* diese als MaxonAsm-Fehlerdatei zu interpretieren und die Fehlermeldungen direkt im Compiler-Fenster anzuzeigen. Also wäre hier der Aufruf `masm sub.asm -c clos -v sub.err` höchst empfehlenwert.

Noch einmal zurück zu den Mehrfach-Anweisungen: Wie gibt man die eigentlich in MCPP ein, da es doch im Compileranweisungs-Dialog für externe Übersetzeraufrufe nur ein Textgadget gibt? Nun, hier sind die Zeilen mit einem Gegenschrägstrich `"\"` zu trennen. Davor und dahinter muß aus Gründen der Eindeutigkeit jeweils ein Leerzeichen stehen.

3.3.5 Weitere Features

Bisher wurden die Grundfunktionen von Make erläutert, die vom internen Make direkt und sichtbar unterstützt werden. Es gibt aber noch eine ganze Menge zusätzlicher Features, die besonders dann von Interesse sind, wenn man seine Projektdatei für den CLI-Compiler *mcppc* von Hand schreibt.

3.3.5.1 Zeilenverbindung

Die Anzahl der Dateien in einem Projekt wächst oft schnell an und mit ihr auch die Längen von Abhängigkeitszeilen oder Linkeraufrufen. Da z. B. Edward die Zeilenlängen auf 256 Zeichen beschränkt, stellt sich irgendwann das Problem, daß man mehr in eine Zeile packen möchte als eigentlich gesund ist.

Die Lösung ist die Zeilenverbindung mit dem Gegenschrägstrich. Endet eine Zeile mit `"\"`, so verbindet Make sie mit der nächsten Zeile. Auf diese Weise können beliebig viele „echte" Zeilen zu einer einzigen Make-Zeile zusammengefaßt werden.

3.3.5.2 Kommentare

Kommentare beginnen für Make mit einem `"#"`.

Beispiel:

```
# makefile für "Demo"
# Version vom 24.05.94
Demo: main.o  # heute nur eine Objektdatei
mcppc main.o -o Demo   # Linkeranweisung
```

Da Kommentare dem Make selbst ziemlich egal sind, werden sie auch nicht irgendwie gemerkt. Aber immerhin merkt sich MaxonC++ Make beim Lesen einer Projektdatei, ob darin Kommentare, Makros oder Regeln (siehe dazu die nächsten Abschnitte) vorkamen. Haben Sie als Benutzer von *MCPP* in der integrierten Umgebung an der Projekdatei etwas verändert, so daß das Makefile neu geschrieben werden muß, dann fragt *MCPP* sicherheitshalber noch einmal nach, ob das offensichtlich von Hand erstellte Makefile wirklich überschrieben werden soll. Es erscheint also eine Dialogbox mit dem Inhalt

Die Projektdatei „<name>“ wurde nicht
von MaxonC++ Make automatisch erstellt.
Wollen Sie sie trotzdem überschreiben?
Ja Nein

Sie haben dadurch bei Bedarf noch Zeit, z. B. eine Shell zu öffnen und Ihr mühsam handgeschriebenes *Makefile* vor der Vernichtung durch *MCPP* zu retten. Sie können die obige Nachfrage natürlich auch verneinen, aber dann wird das Update der Projektdatei nur aufgeschoben, nicht aufgehoben, und bei der nächsten Gelegenheit nervt *MCPP* erneut mit der selben Frage.

3.3.5.3 Makros

Auch einfache Makros können in Make verwendet werden. Dieses Feature geht aber keineswegs so weit wie z. B. die Makros im C-Preprozessor, vielmehr kann man im Text lediglich eine Zeichenfolge (den Makronamen) durch eine andere ersetzen lassen. Aber auch das ist hin und wieder ganz nützlich.

Man definiert sich ein Makro durch eine Zeile der Form

```
<Makroname> = <Inhalt>
```

Als Makroname ist dabei eine beliebige Folge von Buchstaben, Ziffern und dem Unterstrich `"_"` erlaubt.

```
Objects = main.o test.o sub.o
link_2 = mcppc -g40 -gsi -l amiga
X = Exefile
```

Nachdem ein Makro auf diese Weise definiert wurde, kann es benutzt werden. Dazu muß dort, wo das Makro durch seinen zugewiesenen Wert ersetzt werden soll, ein Dollarzeichen `"$"` vorangestellt werden. Ist der Makroname mehr als ein Zeichen lang, muß er außerdem auch noch in runden Klammern eingeschlossen werden. Nach den obigen Definitionen könnte eine Make-Datei also so beginnen:

```
$X: $(Objects)
$(link_2)
$(Objects)
```

Daran sehen Sie auch schon die beiden typischen Anwendungen von Makros im Makefiles: Zum einen spart man bei geeigneter Anwendung Tipparbeit, zum anderen muß man für bestimmte Änderungen wie das Hinzufügen weiterer Objektdateien nur noch an einer Stelle der Datei die Makrodefinition verändern, statt an mehreren Stellen den gleichen Mist einzutippen. Nebenbei räumen geschickt gewählte Makros den Quelltext auf und sorgen so für mehr Übersicht.

Es gibt einige vordefinierte Spezialmakros:

`"$$"` ist fest mit dem Dollarzeichen belegt. Wenn Sie an irgend einer Stelle Ihrer Projektdatei ein einzelnes `"$"` haben sollen, z. B. als Bestandteil eines eigentümlichen Dateinamens wie `"a$b"`, müssen Sie also ein doppeltes Dollarzeichen schreiben (`"a$$b"`), um Make davon abzuhalten, das nachfolgende (hier also `"b"`) als Makro zu behandeln. Außerdem gibt es noch `"$*"` und `"$@"`, die bei Regeln eine wichtige Rolle spielen:

3.3.5.4 Regeln

Es ist auf die Dauer ziemlich ermüdend, immer wieder von Hand einzutippen, daß Make aus einer auf *„.asm"* endenden Textdatei eine *„.o"*-Objektdatei erzeugen soll, indem es einem bestimmten Assembler mit bestimmten Optionen aufruft. Dieser Vorgang kann durch Make-Regeln erheblich abgekürzt werden.

Die Definition einer Regel sieht auf den ersten Blick wie ein ganz normaler Makefile-Eintrag aus, nur daß der Name der (Pseudo-)Zieldatei aus den Endungen von Quell- und Zieldatei zusammengesetzt ist und keine Abhängigkeiten angegeben werden dürfen. Eine typische Regel für Assembler-Module könnte z. B. so aussehen:

```
.asm.o:     masm -c clos $*.asm -o $@
```

Das Makro `"$*"` wird durch den seiner Endung beraubten Dateinamen, `"$@"` durch den vollständigen Namen der Objektdatei ersetzt.

MaxonC++ Make wendet Regeln nur an, um Objektdateien zu erzeugen, die in der Abhängigkeitszeile der Linkeranweisung genannt werden und für die es keinen Eintrag mit einer expliziten Compileranweisung gibt. Dann wird nach der ersten Regel gesucht, bei der die Endung der Zieldatei (meist `".o"`) stimmt und eine entsprechende Quelldatei auch existiert.

Beispiel:

```
.cc.o: mcppc -c -pp -O $*.cc
.c.o:  mcppc -c -pc -O $*.cc
Demo:  main.o test.o
   mcppc main.o test.o sub.o -o Demo
main.o:main.c test.h
```

Wenn nun die Dateien *„main.c"*, *„main.cc"*, *„test.cc"* und *„sub.cc"* existieren, wird für *„test.o"* die erste, für *„main.o"* die zweite und für *„sub.o"* überhaupt keine Regel benutzt - denn wenn *„sub.o"* nicht in der Abhängigkeitszeile der Exe-Datei auftaucht, sieht Make auch keinen Handlungsbedarf zur Erzeugung derselben.

Für *„main.o"* wurde im Beispiel sogar eine Abhängigkeitszeile eingeführt. Gibt es eine solche nicht, setzt Make automatisch den Namen der Quelldatei als einzige Abhängigkeit.

Es gibt beim MaxonC++ Make zwei eingebaute Regeln:

`.c.o: mcppc -c $*.c` übersetzt C- oder C++-Quelltexte. Beachten Sie, daß MaxonC++ Make bei interner Übersetzung als Compiler-Optionen die global eingestellten Defaults verwendet. Außerdem gibt es eine `".asm.o"`-Anweisung für Assembler-Module. In der integrierten Umgebung *MCPP* kann die zugehörige Anweisungszeile im System-Dialogfenster eingegeben werden (übrigens die selbe Anweisung, die auch benutzt wird, wenn der Menüpunkt **„Text und Objekt aufnehmen"** auf eine *„.asm"*-Datei angewendet wird), beim *mcppc* ist sie belegt mit: `.asm.o: masm $(AFLAGS) $*.asm`

Wenn Sie mit dem *mcppc* arbeiten, sollten Sie folglich dem Makro `"AFLAGS"` die passenden Optionen zuweisen, z. B. `"-c clo"`.

4. Der Linker

4.1 Prinzipielles

Ein Linker erfüllt im wesentlichen zwei Aufgaben:

☞ Er fügt Module eines Programms zusammen. Das ist ein nahezu unverzichtbares Feature, denn ab einer gewissen Größe ist es praktisch unmöglich, ein Programm in einem einzigen Modul zu schreiben.

☞ Anschließend bindet er vordefinierte Bibliotheksroutinen in das Programm ein. Auch das ist eine Aufgabe, die aus dem Programmieralltag nicht wegzudenken ist, denn jeder C- oder C++-Programmierer erwartet ganz selbstverständlich, daß simple Funktionen wie `"strcpy"` oder aufwendige Operationen wie `"printf"` in irgendwelchen Bibliotheken vorhanden sind und vom Linker anstandslos eingebunden werden.

Diese beiden Funktionen überschneiden sich allerdings, denn der Unterschied besteht einzig und allein darin, daß man dem Linker ausdrücklich sagt, welche Module er in ein Programm einbinden soll, während er die benutzten Bibliotheksfunktionen weitgehend selbständig hinzulinkt - und, wenn er hinreichend dumm ist, auch noch erheblich mehr - vor allem im Workstation-Bereich gibt es Compiler-/Linker-Systeme, bei denen ein **"Hello, World!"**-Programm reichlich 50 KByte groß ist, weil der Linker gnadenlos riesige Bibliotheken komplett einbindet.

Das ist es, was man allgemein von einem Linker erwartet, zumal die heute gebräuchlichen Linker immer noch auf Konzepten der 50er Jahre basieren, wie Stroustrup einmal sagte. Auch der Standard-Linker des Amiga, "*ALink*", die verbesserte Version "*BLink*" oder die diversen Nachbauten dieser Linker tun im großen und ganzen nichts anderes. Auch der integrierte Linker von MaxonC++, der zu ALink/BLink kompatibel ist, beherrscht diese beiden Aufgaben, wenn auch zum Teil etwas eigenwillig. Nun reden wir hier aber von C++ und nicht von solchen Antiquitäten wie Fortran oder C, und die neue Zeit fordert nicht nur neue Compiler, sondern stellt auch neue Anforderungen an die damit verbundenen Linker.

Eine der zahlreichen, auf den ersten Blick eher unscheinbaren Neuerungen, die C++ gegenüber C so auf dem Kasten hat, ist die Initialisierung statischer Daten. In C dürfen globale Variablen ausschließlich mit konstanten Werten initialisiert werden, weshalb eine Deklaration wie

```
int i = 42;
```

weder Compiler noch Linker vor unlösbare Probleme stellt: Die Variable `"i"` wird im Datenhunk des Programms angelegt, und die entsprechende Speicherstelle bekommt von Anfang an den Wert `"42"`. Schon in dem Augenblick, wenn das Programm in den Speicher geladen wird, steht also an der Stelle, an der sich später einmal die Variable `"i"` befinden wird, das entsprechende Langwort, und das Schönste daran ist, daß das den Linker überhaupt nicht zu interessieren braucht, denn er bekommt auf diese Weise gar nicht mit, daß es eine solche Initialisierung gibt.

In C++ ist die ganze Angelegenheit schon etwas fortschrittlicher, denn dem Programmierer werden in seinen Initialisierungsdrang keinerlei Zügel mehr angelegt. Das folgende ist zum Beispiel ein korrektes C++-Programm:

```
#include <stdio.h>

int murx = printf("Sachen gibt\'s!!!");

void main() { }
```

Obwohl das Hauptprogramm **"main"** scheinbar leer ist, macht dieses Programm etwas, denn noch vor der Ausführung von **"main"** wird die globale Variable **"murx"** mit dem Ergebnis eines **"printf"**-Aufrufs initailisiert (das Ergebnis von **"printf"** ist die Anzahl der ausgegebenen Zeichen oder ein Fehlercode). Also passiert hier schon eine Ausgabe, bevor überhaut das scheinbare Hauptprogramm gestarten wird.

In dieser Form ist das natürlich ein eher krankhafter Fall von Programmverschleierung, aber durch Konstruktoren kommt es in C++ oft zu solchen absolut unkonstanten Initialisierungen. Die ganz besondere Pointe ist, daß jedes Modul solche Initialisierungen besitzen kann. Lange Rede, schwacher Sinn: Der Linker muß in C++ nicht bloß linken, sondern auch dafür sorgen, daß Module korrekt initialisiert werden - ja, nicht nur das: falls globale Variablen einen Destruktor haben, muß er sogar dafür sorgen, daß nach Programmende derartige Destruktoren aufgerufen werden.

Sie werden lachen, aber der MaxonLINKER kann das! Wie das genau vor sich geht, wird an anderer Stelle erläutert. Kommen wir statt dessem noch einmal auf die beiden anfangs erwähnten Aufgaben des Linkers zu sprechen.Wie Sie MaxonC++ dazu bringen, Module zusammenzubinden, dürfte aus den vorhergehenden Kapiteln hoffentlich klargeworden sein (Nämlich in der integrierten Umgebung mit einem Makefile und beim "*mcppc*" wahlweise mit einem Makefile oder entsprechenden Kommandozeilen-Argumenten). Deshalb widmen die beiden folgenden Abschnitte sich der anderen Funktion, dem Einbinden von Bibliotheksfunktionen:

4.2 Standard-Funktionen und das "Logfile"

Der Linker von MaxonC++ beherrscht das Einbinden von Bibliotheksroutinen erstaunlicherweise gleich auf zwei verschiedene Arten. In diesem Abschnitt soll von der etwas unkonventionelleren Methode die Rede sein, nämlich dem Linken mit Objekt-Dateien über eine Datei namens "*Logfile*".

Im großen und ganzen ist eine Bibliothek nichts anderes als eine Sammlung von Modulen mit nützlichen Funktionen. Der Linker geht nun hin und nimmt sich selbsttätig aus dieser Sammlung alle Module, die gerade benötigt werden, und benötigt werden eben jene Objekt-Dateien, in denen Symbole (also z. B. Funktionsnamen) deklariert werden, die im aktuellen Programm benutzt werden, bisher aber nicht definiert wurden.

Es ist deshalb ein naheliegender Gedanke, dem Linker in Form einer Datei mitzuteilen, welche Objektdatei er einlinken soll, wenn ein bestimmtes Symbol sonst nicht bekannt ist. Genau dieses genial einfache, aber zugegebenermassen nicht sehr kompatible Prinzip wurde bei MaxonC++ imple-

mentiert: Im "*lib*"-Verzeichnis, in dem sämtliche Linker-Libraries stehen, gibt es eine Datei namens "*logfile*", welche Informationen darüber enthält, was wo steht. Der Dateiname wurde zugegebenermaßen etwas eigenartig gewählt, denn normalerweise versteht man unter einem "*log*" so etwas wie ein Protokoll - jeder TeX-User kennt die "*.log*"-Files. Hier ist das "*log*" ganz einfach eine legasthenische Verkürzung von "Katalog". Wenn ich ehrlich sein soll, habe ich das "*logfile*" so genannt, weil das einfach gut klingt und mir gerade nichts besseres einfiel. Wesentlich interessanter dürfte für Sie sein, was in dieser Datei enthalten ist.

Das "*logfile*" ist eine ganz gewöhnliche Textdatei, die man leicht mit jedem Editor bearbeiten kann, und besteht aus einer Folge von Einträgen wie

```
Dateiname1
   Symbol1
   Symbol2
   Symbol3
Dateiname2
   Symbol4
usw.
```

Eine Zeichenfolge, die in der ersten Spalte beginnt, stellt also den Namen einer Objektdatei dar, die sich nach Möglichkeit ebenfalls im "*libs*"-Verzeichnis befinden sollte. Falls jedoch im Dateinamen ein `":"` oder `"/"` vorkommen sollte, kann die Objektdatei liegen, wo sie will, Hauptsache der Pfad wurde auch richtig angegeben.

Jedem Dateinamen kann eine nahezu beliebige Folge von Symbolnamen folgen, die sich dadurch auszuzeichnen haben, daß sie in ihrer Zeile um mindestens ein Leerzeichen eingerückt sind. In diesem Zusammenhang sei davor gewarnt, hier <TAB>´s zu benutzen, denn die mag der Linker überhaupt nicht.

Zur Steigerung der Anschauung hier ein Auszug aus dem originalen Logfile:

```
startup.o
   __StartUp
dosbase.o
   exit_i
   _exit
   abort_standard
   _DosBase
   _SysBase
   _Filehandle_Std_Input
   _Filehandle_Std_Output
   _cin
   _cout
   _cerr
   _std__in
   _std__out
   _std__err
   _errno
   __Writefile
   __Writefile__NoBuf
```

```
wbstartup.o
   _wbmain
stdinput.o
   _getc
   getc__P06stream
   _fgetc
   fgetc__P06stream
   _getchar
   getchar_
   __Readfile
   __Readfile__NoBuf
arg.o
   main_
```

Es gibt hier einige (übrigens in Assembler geschriebene) Objektdateien, in denen bestimmte Linker-Symbole deklariert werden, zumindens behauptet das Logfile das. Wenn also in einem Programm ein Symbol namens `"_getc"` referiert wird, weiß der Linker, daß er die Datei "*stdinput.o*" aus dem "*libs*"-Verzeichnis einbinden soll.

Es gibt im wesentlichen für jede Funktion eine eigene Objektdatei. Oft wurden aber aus rein praktischen Erwägungen Funktionen mit gemeinsamen Unterroutinen (z. B. `"getc"` und `"fgetc"`) oder sinngemäß zusammenhängende Funktionen (kein `"fopen"` ohne `"fclose"`) in einer Objektdatei zusammengefaßt. Effektiv wird hier also nur das eingebunden, was auch tatsächlich benutzt wird.

Damit dürfte auch klar sein, wie Sie selbst die Standard-Bibliotheken erweitern können: Kopieren Sie einfach die Objektdatei in das "*libs*"-Directory, und fügen Sie dem Logfile einen entsprechenden Eintrag hinzu. Wenn Sie sich nicht sicher sind, wie die Linkersymbole zu ihren Routinen heißen - was in C++ ja nicht so ganz trivial ist, da ja die Parameterlisten in den Symbolnamen hineincodiert werden, gibt es zwei Möglichkeiten, diese herauszufinden: Entweder lassen Sie sich ein Assembler-Listing erzeugen und suchen sich dort die `"XDEF"`-Zeilen heraus, oder Sie versuchen zuerst einmal "einfach so", die selbstgeschriebenen Routinen einzubinden, und merken sich dann, was der Linker nicht finden kann.

4.3 Das Hauptprogramm und seine Argumente

Wenn Sie sich das Logfile durchlesen, erweckt vielleicht der Eintrag `"main_"` für die Datei "*arg.o*" Ihr berechtigtes Interesse, auch wenn dieser leider eben nicht besagt, daß man das Hauptprogramm überhaupt nicht mehr selbst schreiben muß. `"main_"` ist in C++-Linkage der Name des parameterlosen Hauptprogramms. Nun darf man `"main"` natürlich auch mit den beiden kanonischen Argumenten deklarieren, also so:

```
int main(int, char**)
```

Wird dies im C++-Modus übersetzt, lautet das zugehörige Linkersymbol `"main_iPPc"`, und im C-Modus wird der Programmname für den Linker immer in `"_main"` übersetzt.

Dieses Dilemma löst MaxonC++ elegant dadurch, daß der Startup-Code zunächst in eine Funktion `"main_"` einspringt. Wenn der Linker dieses Symbol nicht findet, liegt das entweder daran, daß der Programmierer das Hauptprogramm schlicht vergessen hat, oder es ist eben darauf zurückzuführen, daß `"main"` mit Parametern oder im C-Modus deklariert wurde. Dem Linker ist dies zunächst einerlei, und da das Symbol `"main_"` im Startup-Code referenziert, aber nirgendwo definiert wird, sucht er es wie jedes andere Symbol in den Bibliotheken. Dort wird er auch prompt fündig und bindet die Objektdatei "*arg.o*" ein. Hinter dem Symbol `"main_"` verbirgt sich in diesem Bibliotheksmodul eine kleine Routine, die den Argumentvektor `"argv"` erzeugt und initialisiert, `"argc"` und den Zeiger auf `"argv"` auf den Stack ablegt und dann in eine Routine `"main_iPPc"` einspringt.

Wurde das Hauptprogramm im C++-Modus mit den üblichen Parametern deklariert, trägt es eben jenes Symbol, und die Sache ist für den Linker gegessen. Im C-Modus muß er noch einen weiteren Schritt machen: er bindet zusätzlich das Modul `"another_main.o"` ein, in dem ein Symbol `"main_iPPc"` deklariert wird. In dieser Mini-Datei steht nichts anderes als ein Sprung an eine Adresse namens `"_main"`, und auf diese Art und Weise wird auch in C-Programme korrekt eingesprungen.

Das Angenehme an dieser Rumlinkerei ist, daß auf diese Weise der Code für die Berechnung von `"argv"` in C++ nur dann eingebunden wird, wenn das Programm die Argumente auch auswertet, das spart einige Bytes an Codegröße. Im C-Modus wird dieser Code leider immer eingebunden, und hinzu kommen noch 8 Bytes sowie ein Relokationstabelleneintrag für den zusätzlichen Sprung.

Ach ja, noch was: Es gibt auch eine Bibliotheksfunktion `"_wbmain"`. Wenn Sie also nicht selbst eine Startup-Routine für die Workbench schreiben, nimmt der Linker die aus der Bibliothek. Diese Funktion macht aber nichts anderes als umgehend auszusteigen. MaxonC++ geht davon aus, daß Programme, die nicht eigens dafür geschrieben sind, von der Workbench gestartet zu werden, nicht künstlich Workbench-fähig gemacht werden sollten (etwa durch zwangsweises Öffnen eines Text-Fensters oder ähnlichen Schnickschnack). Daß ein MaxonC++-Programm also nicht ohne weiteres von der Workbench startbar ist, ist kein Bug, sondern ein Feature.

Näheres zu der Frage, wie man ein Programm Workbench-tauglich macht, steht übrigens im Kapitel 6.2 des Tutorials.

4.4 Linker-Bibliotheken

Ich glaube, ich erwähnte schon, daß die Sache mit dem Logfile eine Spezialität von MaxonC++ ist. Der große Vorteil ist dabei, neben der einfachen und flexiblen Erweiterbarkeit, die recht hohe Geschwindigkeit vor allem bei langsameren Speichermedien: Der Linker hat detaillierte Informationen darüber, was er einbinden soll, und muß dann nur noch die benötigten Dateien laden.

Gebräuchlicher sind hingegen Linker-Libraries, bei denen die gesamte Bibliothek in einer Datei zusammengefaßt wird. Der Linker liest sich diese Datei aufmerksam durch und pickt sich die benötigten Teile heraus. Interessant ist dieses Verfahren vor allem deshalb, weil die offiziell von Commodore vertriebene "*amiga.lib*" in einem solchen standardisierten Format vorliegt - natürlich alles "ALink/BLink"-kompatibel.

Aus diesem Grund unterstützt auch der MaxonC++ Linker auch dieses Bibliotheksformat. Man kann eine beliebige Anzahl solcher Bibliotheken angeben, aus denen er dann alles nimmt, was er sonst nicht kriegen kann.

Rein technisch erfolgt die Deklaration der zu untersuchenden Bibliotheken in der Kommandozeilenversion mit der Option `"-l"` und in der integrierten Umgebung mit einem entsprechenden Eintrag im entsprechenden Symbol des Linker-Dialogfensters. Auch in der Makedatei kann man die gewünschten Bibliotheken in der zweiten Zeile (der Linker-Anweisung, die mit "*mcppc*" anfängt) angeben.

4.5 Initialisierungen und das große Aufräumen

In 4.1 wurde schon mit einem gewissen Stolz darauf hingewiesen, daß der MaxonC++ Linker als extraordinäres Feature auch Module initialisieren kann. Da fragt sich der geneigte Leser natürlich, woher dieses Wunderprogramm weiß, was wie initialisiert werden soll.

Die Lösung dieses Problems ist ganz einfach: Wenn der Linker ein Symbol findet, das mit `"_INIT_x_"` anfängt - wobei `"x"` eine Ziffer zwischen `0` und `9` ist - weiß er, daß diese Funktion im Startup-Code aufgerufen werden soll. Umgekehrt werden Funktionen mit dem Anfang `"_EXIT_x_"` am Programmende aufgerufen. Die Ziffer `"x"` gibt in beiden Fällen die Priorität der Initialisierung bzw. des Cleanups an: Initialisierungen mit der Nummer `0` werden als erste aufgerufen, die mit `9` als letzte. Bei den

`"_EXIT_"`-Funktionen ist die Reihenfolge umgekehrt. Wenn der Compiler ein Modul erzeugt, das initialisiert werden muß, erzeugt er eine Initialisierungsfunktion mit der Priorität 8, und für eventuelle Destruktoraufrufe wird entsprechend eine Exit-Funktion gleichen Vorrangs generiert.

Der genaue Name der Init- und Exit-Symbole ist belanglos, sie sollten nur im Programm möglichst eindeutig sein. Deshalb kodiert der Compiler den Namen der Objektdatei in die Funktionsnamen, während in den fertigen Libraries (Sie erinnern sich: alle in Assembler geschrieben) möglichst seltsame Namen gewählt wurden, die bestimmt nirgendwo sonst vorkommen, z. B. `"_INIT_0_files_o_Yeah_Initialise_me_Babe"` im Modul "*files.o*".

Der Linker erzeugt selbstständig einen kleinen Startup-Code, der erst sämtliche Initialisierungen in der Reihenfolge ihrer Priorität aufruft und dann in den eigentlichen Startup-Code, der für gewöhnlich aus dem Bibliothksmodul "*startup.o*" eingebunden wird, einspringt. Außerdem generiert der Linker stets eine Funktion, in der sämtliche Exit-Funktionen aufgerufen werden. Die Standard-Funktion `"exit"` springt dieses Label an.

4.6 Mit und ohne Startup-Code

Manchmal will man den oben beschriebenen Startup-Code nicht haben, zum Beispiel, weil man eine Shared Library programmieren will oder weil einem der Startup-Code einfach nicht paßt. Deshalb gibt es einige Linkeroptionen, die eben diesen unterdrücken.

Zuerst wäre da die schlichte *CPPC*-Option `"-gs"` oder die Einstellung "***Ohne Startup-Code***" im "**Linker**"-Dialogfenster von "*MCPP*". Bei dieser Option werden die Module des Programms in der angegebenen Reihenfolge gelinkt, die benötigten Bibliotheksroutinen werden angehängt und als allerletzter Hunk wird eine Sparversion des Startup-Codes erzeugt. Dieser kleine Hunk enthält dann zwei parameterlose Funktionen mit den Labels `"_InitModules"` und `"_CleanupModules"`. Diese sollte man auf jeden Fall am Programmanfang bzw. -ende aufrufen. Von C++ aus muß man sie natürlich in C-Linkage importieren, z. B. so:

```
extern "C"
{ void InitModules();
  void ExitModules();
}
```

Diese Definitionen stehen aber auch im Include-File "*<linkerfunc.h>*".

Falls man, das kanonische Beispiel für Linken ohne Startup-Code, eine Shared Library implementiert, sollte `"InitModules()"` die allererste Anweisung in der Open-Routine der Library und `"CleanupModules()"` entsprechend die letzte Anweisung in der Schließroutine sein.

Für fanatische Bytezähler, aber auch für Leute, die "*mcppc*" als BLink-Ersatz verwenden möchten, hat die CLI-Version noch zwei schlauere Optionen. Zum einen gibt es da noch `"-gsi"`. Hier wird der kleine Zusatz-Hunk mit den Funktionen `"InitModules"` und `"ExitModules"` nur dann erzeugt, wenn eines der beiden Labels auch tatsächlich irgendwo im Programm referiert wird. Dabei sind rätselhafte Programmfehler natürlich schon fast vorprogrammiert, denn man sollte die beiden Funktionen eigentlich vorsichtshalber immer aufrufen, da man ja nie wissen kann, ob irgendwelche Library-Module eine Initialisierung benötigen.

Deshalb sollten Sie `"-gsi"` ganz schnell wieder vergessen und stattdessen die Option `"-gsw"` verwenden. Diese funktioniert ganz genau wie `"-gsi"`, gibt aber eine Warnung aus, wenn eine Initialisierung bzw. ein Cleanup nötig ist, aber nicht durchgeführt wird. Im Klartext: Wenn irgendein Modul eine `"_INIT_x_"`-Funktion besitzt und das Label `"_InitModules"` nirgendwo im Programm referiert wird, gibt der Linker so etwas wie **„_Symbol '_InitModules' never used“** aus - für den Cleanup analog. Die Option `"-gsw"` ist recht praktisch, wenn man sich recht sicher ist, daß nirgendwo im ganzen Programm einschließlich der Standard-Bibliotheken eine dynamische Initialisierung nötig ist, die entsprechenden Funktionsaufrufe deshalb wegläßt und bei Bedarf über seinen Irrtum aufgeklärt werden möchte.

Für echte Freaks ist es natürlich unter ihrer Würde, auf fertige Routinen zurückzugreifen, und so ist ihnen auch der normale Startup-Code suspekt. Es ist nicht weiter schwierig, sich einen völlig eigenen Startup-Code zu schreiben. Linkt man ein Programm ohne Startup-Code und startet es dann,

4.7 Der Startup-Code

Für alle, die genauer wissen möchten, was beim Programmstart passiert, folgt hier eine kommentierte Fassung des Quelltexts für das Bibliotheksmodul "*startup.o*". Wenn man einmal die unvermeidlichen Symbol- und Konstantendefinitionen wegläßt, bleiben ganze drei Funktionen übrig. Hier ist die erste:

```
_INIT_0_Get_StartuP_MessAge:
        move.l sysbase.w,a6
        move.l $114(a6),a5
        tst.l  $AC(a5)
        bne.b  ret
        move.l #__Writeflag,d0
        bne.b  ret
        lea    $5C(a5),a0
        move.l a0,-(a7)
        jsr    WaitPort(a6)
        move.l (a7)+,a0
        jsr    GetMsg(a6)
        move.l d0,StartupMessage
ret     rts
```

Diese Init-Routine mit Priorität 0 ist garantiert das erste Stück Code, das im Programm ausgeführt wird - es sei denn, ein heimtückischer Assembler-Programmierer fügt eine Init-Routine gleicher Priorität hinzu. Wie der Amiga-Kenner leicht sieht, wird hier als erstes geprüft, ob das Programm von der Workbench oder aus der Shell gestartet wurde. Trifft ersteres zu, wird die Startup-Message abgeholt und in einer Variablen namens `"StartupMessage"` abgelegt.

Das seltsame Symbol namens `"__Writeflag"` ist übrigens ein übler Hack, durch den die Umgebung *MCPP* Zugriff auf die Standard-Ein- und -Ausgabe des Programms erhält. *MCPP* trägt dort die Adresse einer trickreichen Routine ein, während es bei einem "richtig" gelinkten Programm immer Null ist. Daß dieser Wert in der obigen Routine abgefragt wird, war bei antiquierten Versionen von *MCPP* noch nötig. In absehbarer Zeit wird dieser Test wohl aus dem Startup-Code 'rausfliegen.

Der kurze Code, den der Linker im normalen Modus an den Anfang des Programms setzt, ruft also zuerst alle Init-Routinen in der richtigen Reihenfolge auf und springt dann in die Routine `"__Startup"` ein. Wenn man sie nicht überdefiniert, nimmt er dafür die folgende:

```
__StartUp:
        move.l StartupMessage,d0
        beq.b  .cli
        move.l d0,-(a7)
        jsr    _wbmain
        addq.l #4,a7
        moveq  #0,d0
        rts
.cli    jmp    main_
```

wird in die erste Routine in der ersten Übersetzungseinheit eingesprungen. Es kommt dann also beim Linken wirklich auf die Reihenfolge der Module an. Durch geschickt definierte Registerparameter kann eine solche Funktion sogar die Shell-Argumente auswerten. Zuvor sollte sie aber die Module initialisieren und im Zweifelsfall mit `"GetBaseReg()"` den Zeiger auf den Small-Data-Hunk initialisieren.

Das Beispielprogramm "*nostartup.c*" demonstriert dies:

```
// nostartup.c - Demo für Linken ohne Startup-Code//
// Compileraufruf: cppc nostartup.c -gs
// bzw. "Ohne Startup" im M+Plus-Linker-Requester einstellen

#include <linkerfunc.h>
#include <stream.h>

// Die erste Funktion im ersten Hunk wird angesprungen:
void f(register __d0 int len, register __a0 char *par)
// In Registern d0 und a0 werden die Shell-Parameter übergeben
// Diese Register-Belegungen bremsen die Funktion übrigens gut aus,
// deshalb sollte man in dieser Funktion nicht mehr als nötig tun.
{
  // Wichtig, wenn Small Data-Hunks existieren, und sonst auch
  // nicht verkehrt:

  GetBaseReg();    // Initialisierungen, z. B. Dos-Lib öffnen:
  InitModules();
  par[len-1]=0;       // Normalerweise kein
                        // Nullbyte am Ende
  // Jetzt geht alles wie immer:
  cout << "Es geht auch ohne Startup-Code!" <<
  "\nParameter-Länge: " << len << "\nParameter: " << par
  << "\n";
  // Programmende: Aufräumen, z. B. DosLibrary schlie_en:
  CleanupModules();

  // "exit()" darf ohne Startup-Code nicht benutzt werden!!!
  // Es ist nur so ein sinnvoller Return-Code möglich:
  len = 42;
}
```

Es handelt sich also um eine simple Fallunterscheidung, die entweder in `"main_"` oder `"_wbmain"` einspringt. Diese beiden Funktionen wurden schon in Abschnitt 4.3 besprochen.

So, jetzt bleibt nur noch eine kurze Funktion:

```
_EXIT_0__Reply_StartuP_MessAge:
        move.l  StartupMessage,d2
        beq.b   ret
        move.l  sysbase.w,a6
        jsr     Forbid(a6)
        move.l  d2,a1
        jmp     ReplyMsg(a6)
```

Durch die Priorität 0 ist dies die letzte Funktion, die am Programmende aufgerufen wird. Wurde das Programm von der Workbench gestartet, wird hier die Startup-Message beantwortet, und das wars dann schon.

Assembler-Programmierer können aus diesen Listings auch noch die Erkenntnis gewinnen, daß Init- und Exit-Funktionen offensichtlich keine Rücksicht auf das Trashing von Registerwerten nehmen müssen.

4.8 Hunks von Aufteilen, Zusammenfassen und Wegschmeißen

An dieser Stelle soll noch einmal an drei besondere Features des MaxonC++ Linkers erinnert werden - nur damit Sie sich nicht wundern, wenn Sie die Hunks Ihrer Objektdateien im ausführbaren Programm nicht mehr so wiederfinden, wie Sie es vielleicht erwartet haben.

Zum einen ist da die Eigenschaft, daß unreferenzierte Hunks grundsätzlich nicht in die Exe-Datei aufgenommen werden. Dies erwies sich in der Version 3.0 als nahezu unverzichtbar, da bei Templates "auf Verdacht" jede Menge Code angelegt werden muß, von dem am Ende vieles nicht gebraucht wird. Obendrein werden diese Funktionen oft in mehreren Übersetzungseinheiten zugleich angelegt, was ja nun wirklich nicht wein muß. Deshalb wurde eine Konvention eingeführt: Alles, was in einer Objektdatei ab dem zweiten Codehunk steht, wird wie ein Library-Modul behandelt, d. h. Überschneidungen mit identischen Symbolen aus anderen Objektdateien werden ignoriert.

Dieses großzügige Hunk-Wegwerfen war Voraussetzung für den Library-Modus, der strenggenommen ein Compiler-Feature ist. Wie schon in Abschnitt 2.5.3.2 beschrieben, wird bei der Codeerzeugung für jede Funktion und jede globale Variable ein eigener "Hunk" erzeugt. Der Linker schmeißt kann nicht benutzte Funktionen platzsparenderweise heraus.

Und zu guter letzt sei noch einmal an das optionale Zusammenfassen von Hunks erinnert (2.5.3.5). Dadurch sehen die ausführbaren Programmdateien wesentlich aufgeräumter aus, und ein paar Bytes spart es auch.

5. Die CLI-Version "MCPPC"

5.1 Was ist neu?

Wenn Sie bereits die 1.x-Version von Maxon C++ kennen, wird Sie als erstes sicher interessieren, was das vorliegende Update Neues bringt. Sehen wir einmal davon ab, daß alles besser und schöner geworden ist, so kommen wir auf folgende Feature-Liste:

- Bei Make-Dateien ist natürlich das in Abschnitt 3.3 vorgestellte, erweiterte Format erlaubt.
- In den Make-Dateien können die Abhängigkeiten vollautomatisch gesetzt werden.
  Optionen `"-mx"` und `"-mX"`
- Es gibt drei verschiedene Formate bzw. Modi für Fehlerausgaben: mit `"-ei"` schaltet man in den bekannten interaktiven Modus (mit Nachfrage nach jedem Error), `"-e<N>"` ist wie gehabt die Option für die Kurzausgabe und mit `"-el<N>"` erreicht man das neue Langformat. Die optionale Angabe `"<N>"` steht dabei für die maximale Anzahl von Meldungen.
- Der Quelltext bzw. die Positionsangabe können zwecks besserer Übersichtlichkeit in beliebiger Farbe ausgegeben werden.
  Option `"-ec<N>"`
- Die eigentliche Fehlermeldung kann auf Wunsch fett („bold") hervorgehoben werden.
  Optionen `"-eb"` und `"-eB"`
- Fehlermeldungen können in einer Datei abgelegt werden.
  Option `"-ef"`
- Warnungen können wie Fehler behandelt und leichter eingeschaltet werden.
  Optionen `"-ew"` bzw. `"-w+"`
- Das Exception Handling (also eines der wesentlichen neuen Compiler-Features) kann ein- oder ausgeschaltet werden.
  Optionen `"-px"` und `"-pX"`
- Beim Erzeugen von Template-Funktionen sind zwei verschiedene Strategien möglich.
  Optionen `"-t"` und `"-T"`
- Code für die 32Bit-Prozessoren (ab 68020) und Coprozessoren kann erzeugt werden.
  Optionen `"-g00"` und `"-gs0"` bzw. `"-g80"` und `"-g81"`
- Echt vorcompilierte Header-Dateien können gelesen und abgespeichert werden.
  Optionen `"-h0"`, `"-h <Name>"` und `"-H <name>"`
- Im Library-Modus wird für jede Funktion und jede Variable ein separater Hunk erzeugt.
  Optionen `"-gl"` und `"-gL"`

☞ Zu jedem Switch gibt es jetzt eine Option, mit der man explizit die Default-Einstellung einschalten kann.

☞ Für einige wichtige Optionen wurden Abkürzungen eingeführt:

`-O`	`-gO`	Optimieren
`-L`	`-gL`	Library-Modus
`-X`	`-pX`	Ausnahmebehandlung zulassen

5.2 Das Konzept

Es soll angeblich tatsächlich Leute geben, denen dir integrierte Entwicklungsumgebung von MaxonC++ nicht gefällt. Deshalb gibt es eine CLI-Version von MaxonC++, nämlich das Programm "MCPPC".

Eigentlich ist MCPPC ebenfalls ein integriertes System, denn es enthält Compiler, Assembler und Linker. Der Unterschied zum "MCPP" liegt also ausschließlich in der Benutzerschnittstelle: MCPPC funktioniert wie ein Shell-Befehl und wird über Kommandozeilen-Argumente gesteuert.

Das System wurde ähnlich wie die C-Compiler unter Unix konzipiert, d. h. compilieren, assemblieren und linken erfolgt mit einer einzigen Anweisung (oder optional auch getrennt). Dateinamen, die auf ".o", ".obj" oder ".l" enden, werden als Objektfiles interpretiert und eingelinkt, alles andere betrachtet der Compiler als Namen von Quelltexten.

Ohne besondere Optionen übersetzt der Compiler alle angegebenen Quelltexte (ohne die entsprechenden ".o"-Files zu schreiben), linkt die angegebenen Objektdateien ein und erzeugt eine ausführbare Datei, deren Name sich vom jeweils ersten Dateinamen ableitet:

```
mcppc test.o haupt.c text.c unter.c nochwas.o
```

compiliert die drei Quelltexte "*haupt.c*", "*text.c*" sowie "*unter.c*", erzeugt aber keine Objektdateien, sondern hält das Ergebnis dieser Übersetzungen resident im Speicher. Anschließend lädt er die beiden Objektdateien "*test.o*" und "*nochwas.o*" dazu und erzeugt daraus eine ausführbare Datei namens "*test*".

Bei kleinen Programmen, die aus nur einem Modul bestehen, ist der Compiler also ausgesprochen einfach zu bedienen: Die Anweisung

```
mcppc name.c
```

übersetzt das Programm, linkt es und erzeugt eine ausführbare Datei namens „*name*". Einfacher geht es wohl nicht.

Tritt bei der Übersetzung ein Fehler auf, wird ein geeigneter kurzer Auszug aus dem Quelltext ausgegeben, zusammen mit einer Ausgabe der Art

```
Datei "Dateiname.c", Zeile 17:
Error: Identifier "ths" not defined.
Weiter mit <Return>, abbrechen mit <A>:
```

Jetzt können Sie also entweder mit der <Return>-Taste den Fehler übergehen und die Übersetzung fortsetzen, oder <A> tasten und damit den Compiler abbrechen.

MCPPC ist residentfähig, auch wenn einige Shells einen Prüfsummenfehler melden. Dies ist in jedem Fall harmlos, auch wenn *MCPPC* auf eine etwas ungewöhnliche Methode reentrant gemacht wurde: Anhand eines Semaphors sorgt eine geladene Kopie des Programms dafür, daß sie nicht zweimal gleichzeitig läuft.

5.3 Optionen

5.3.1 Allgemeines

Die oben beschriebenen Features allein sind natürlich noch etwas mager. Alle darüber hinausgehenden Funktionen werden durch Compileroptionen gesteuert, die in der Kommandozeile angegeben werden und jeweils mit einem Minuszeichen `"-"` beginnen.

Um ein wenig Ordnung in dieses umfangreiche Sammelsurium zu bringen, lassen die Optionen sich anhand ihres jeweils ersten Buchstaben in verschiedene Bereiche einteilen:

- `-a` Assembler-Sourcegenerierung
- `-b` Debugging
- `-c` Wahl zwischen Exe- und Objektdatei
- `-d` Symboldefinitionen
- `-e` Fehlermeldungen
- `-g` Codegenerierung
- `-h` vorcompilierte Header-Dateien
- `-i` Pfade für Includes
- `-j` Pfade für Libraries
- `-l` Linker-Libraries
- `-m` Make
- `-o` Ausgabedatei
- `-p` Parser
- `-w` Warnungen und Workspacegröße

Optionen sind stets global: Bei einer Zeile der Form

```
mcppc -x text1.c -y text2.c -z
```

werden "*text1.c*" und "*text2.c*" gleichermaßen mit den Optionen "*x*", "*y*" und "*z*" compiliert. Mit anderen Worten ist die Reihenfolge von Dateinamen und Optionen egal.

Die Reihenfolge der Optionen untereinander ist aber nicht egal, wenn in einer Anweisungszeile einander widersprechende Optionen angegeben auftreten. Beispielsweise schaltet man mit `"-go"` die Optimierung ein und mit `"-go"` aus. In Fällen wie

```
mcppc -go -g0 usw.etc
```

gilt die jeweils letzte Angabe.

Nicht nur für die Optimierung, sondern für fast jedes Feature gibt es zwei Optionen: eine zum Einschalten und eine zum Ausschalten. Das erscheint auf den ersten Blick überflüssig, ist doch z. B. die

Optimierung standardmäßig ohnehin nicht aktiviert und damit eine explizite Ausschaltoption eigentlich nicht erforderlich. Man kann aber alle diese Grundeinstellungen ändern, und zwar mit der Umgebungsvariablen `"CPPOPT"`, zu der wir später kommen. Wenn Sie damit `"-g0"` zum Default machen, wollen Sie vielleicht doch noch einmal mit `"-go"` zurückschalten.

Bei den meisten Umschaltmöglichkeiten wurde der Weg gewählt, daß man mit einem Großbuchstaben ein- und mit einem Kleinbuchstaben ausschaltet, so wie bei `"-go"` und `"-g0"`. In einigen Fällen, vor allem bei weniger oft benötigten Features, gab es in den 1.x-Versionen von Maxon C++ nur einen Einschalter mit lauter Kleinbuchstaben und keine explizite Rückschaltmöglichkeit. Das wichtigste Beispiel ist wohl die Option `"-b"` für die Erzeugung von Debugger-Dateien. Aus Gründen der Kompatibilität (wer gewöhnt sich schon gerne um, vor allem bei solchen ohnehin kryptischen Kürzeln) wurden diese Optionen so belassen und die neuen Ausschalter durch Anhängen einer Null definiert. `"-b0"` bedeutet also "Keine Debugger-Dateien schreiben".

In einigen Fällen wurden auch Abkürzungen eingeführt, z. B. ist `"-O"` identisch mit `"-go"`. Okay, eine Zwei-Buchstaben-Kombination durch einen einzelnen Buchstaben abzukürzen, ist vielleicht keine besonders eindrucksvolle Ersparnis. In diesem speziellen Fall hat `"-O"` aber auch die Vorteile, schöner auszusehen und zu einigen anderen Compilern kompatibel zu sein. Was will man mehr?

5.3.2 Ergebnisse und Ausgaben

5.3.2.1 Assembler-Quelltext: -a und -as

Vielleicht möchten Sie sich einmal den von Maxon C++ erzeugten Code ansehen, oder Sie gehören zu den Leuten, die compilierte Programme von Hand nachoptimieren. Mit der Option "-a" können Sie sich dann ein Assemblerlisting ausgeben lassen.

```
mcppc test.c -a
```

erzeugt beispielsweise nicht nur die ausführbare Datei "*test*", sondern auch das Assembler-Listing "*test.s*". Die Angabe `"-a"` bricht den Übersetzungsvorgang also nicht vor der Assemblierung ab, sondern erzeugt den Quelltext zusätzlich zu den übrigen Aktionen.

Ein reines Assemblerlisting ist aber meist ziemlich unübersichtlich. Deshalb gibt es die Variante `"-as"`, durch die Auszüge aus dem originalen Quelltext an den jeweils entsprechenden Stellen als Kommentar in den Assembler-Quelltext eingefügt werden.

Um diese beiden Optionen explizit auszuschalten, verwenden Sie `"-a0"`.

5.3.2.2 Objektdateien: -c

Die Option `"-c"` unterdrückt das Linken (und damit auch die Erzeugung einer ausführbaren Datei) und erzeugt statt dessen eine Objektdatei:

```
mcppc -c file1.c file2.C
```

erzeugt die beiden Objektdateien "*file1.o*" und "*file2.o*" - natürlich nur, wenn die Quelltexte auch erfolgreich übersetzt werdne konnten.

Die Option `"-c"` ist der Schlüssel zur modularen Programmierung: Wenn ein Programm aus einzelnen Übersetzungseinheiten wie "*f1.c*", "*f2.c*" usw. besteht, übersetzt man mit

```
mcppc -c fN.c
```

immer nur die Module, die verändert wurden, und linkt das Ganze anschließend mit

```
mcppc f1.o f2.o f3.o ...
```

zu einer ausführbaren Datei zusammen.

Eher selten wird man dagegen die Option `"-cx"` benutzen: Dabei werden zwar auch ".*o*"-Files erzeugt, aber das Projekt wird trotzdem gelinkt. Und auch hier gibt es einen expliziten Schalter für die Grundeinstellung (also linken und schreiben der ausführbaren Programmdatei), nämlich `"-cX"`.

5.3.2.3 Ausführbare Programmdatei: -o

Wenn der Linker erfolgreich war und eine ausführbare Datei erzeugen konnte, benennt er diese normalerweise nach dem ersten Dateinamen, der in der Anweisungszeile auftrat, indem er einfach dessen Endung wegläßt. Falls das Ihnen nicht gefällt, können Sie mit der Option `"-o"` einen anderen Dateinamen angeben, z. B.

```
mcppc quelle.cpp -o Ergebnis
```

Falls mit `"-c"` auf die Erzeugung von Objektdateien umgeschaltet wurde, legt `"-o"` den Namen der Objektdatei fest.

5.3.3 Preprozessor

5.3.3.1 Definieren von Symbolen: -d

Mit der Option `"-d"` kann man "von außen" ein Preprozessor-Symbol definieren. Dabei entspricht ein Argument

```
-d NAME
```

einer Zeile

```
#define NAME
```

am Anfang des Quelltexts. Eine typische Anwendung ist eine Anweisung wie

```
mcppc text.c -d NDEBUG
```

wenn die Includedatei "*<assert.h>*" benutzt wird.

Man kann dem Symbol aber auch einen Wert zuweisen, indem man unmittelbar hinter den Bezeichner ein `"="` setzt, z. B.

```
-d ANZAHL=4711
```

Als Wert ist hier aber nur ein Token erlaubt, d. h. bei

```
-d N=17+4
```

würde das `"+4"` schlicht ignoriert.

5.3.3.2 Includedateien: -i

Der Pfad, über den CPPC die Includedateien sucht, ist mit "*MCPP:include*" vor-eingestellt. Durch einen oder auch mehrere `"-i"`-Optionen kann dieser Pfad geändert werden: Mit der Option

```
-i MCPP:inc
```

wird ausschließlich im Verzeichnis "*CPP:inc*" gesucht, und bei

```
-i MCPP:include -i MCPP:include/amiga
```

durchsucht der Preprozessor nacheinander die beiden genannten Verzeichnisse.

Tritt in einem Dateinamen ein `"|"` auf, z. B.

```
-i ram:include|df0:include
```

werden die gelesenen Dateien automatisch vom zweiten in den ersten Pfad umkopiert und dabei in ein kompakteres und schneller übersetzbares Token-Format umgewandelt, wie es am Ende von Abschnitt 2.5.2.1 beschrieben wurde.

5.3.4 Compiler-Modus

5.3.4.1 ANSI C oder C++: -pc und -pp

Maxon C++ kann durchaus auch als ANSI C-Compiler verwendet werden. Wenn Sie dazu nicht ein `"#pragma -"` an den Anfang des Quelltexts setzen wollen, können sie auch die Compileroption `"-pc"` verwenden. Default ist natürlich der C++-Modus, dem die Option `"-pp"` entspricht. Also `"c"` wie `"C"` und `"p"` wie `"Plus Plus"` - ist doch ganz leicht zu merken.

5.3.4.2 Exception Handling: -px, -pX und -X

Wie in Kapitel 8.5 des Tutorials dargestellt, hat es manchmal Vorteile, wenn man ein "klassisches" C++-Programm, das kein Exception Handling benutzt, in einem speziellen Compilermodus überset-

zen läßt. Dann werden die mit der Ausnahmebehandlung zusammenhängenden Schlüsselwörter (`"catch"`, `"throw"` und `"try"`) nicht mehr erkannt, und im erzeugten Code wird nicht über notwendige Destruktoraufrufe Buch geführt. Das bringt in C++-Programmen eine leichte Verbesserung in Bezug auf Größe und Geschwindigkeit, und bei ANSI C-Programmen hat es sowieso keine Auswirkung.

Als Default ist das Exception Handling bei *MCPPC3* ausgeschaltet, was der Option `"-px"` entspricht. Wenn Sie die Ausnahmebehandlung benutzen möchten, müssen Sie sie erst mit `"-pX"` einschalten. Als Abkürzung ist auch `"-X"` erlaubt.

5.3.4.3 Erzeugen von Template-Funktionen: -t und -T

Wie in Abschnitt 2.5.2.4.10 ausführlich dargestellt, ist es für den Compiler nicht immer leicht zu entscheiden, welche Template-Funktionen er erzeugen soll und welche nicht. `"-T"` entspricht der Strategie "alle Templates erzeugen" bzw. dem angewählten Schalter "***Alle Templates***" im "Compiler"-Dialogfenster, bei `"-t"` wird die andere Strategie verfolgt, also nur das erzeugt, was sich fehlerfrei erzeugen läßt.

5.3.5 Codeerzeugung

5.3.5.1 Breakpoints setzen: -gb

Wenn die Option `"-gb"` benutzt wird, setzt der Compiler in alle Schleifen und sonstige schleifenverdächtige Konstrukte eine Abfrage des Break-Bits, das z. B. bei Betätigung der Tastenkombination <Ctrl>+<C> gesetzt wird. Dadurch ist es möglich, so erzeugte Programme jederzeit bequem abzubrechen, allerdings kostet diese Option auch einiges an Laufzeit.

Der zugehörige Ausschalter ist `"-gb0"`.

5.3.5.2 Datenmodell: -gd und -gD

Die Option `"-gd"` schaltet auf "Small Data", `"-gD"` auf "Large Data" (siehe 2.5.3.1). "Large" ist Default.

5.3.5.3 Zusammenfassen von Hunks: -gh

Mit der Option `"-gh"` legen Sie die Höchstgrenze in KByte fest, bis zu welcher der Linker gleichartige Hunks verbindet (siehe auch 2.5.3.5), zum Beispiel: `"-gh20"` für 20 kByte. Default ist 50.

5.3.5.4 Ganzzahlige Arithmetik: -gm

Wie bereits unter 2.5.2.4.3 beschrieben, hat der 68000-Prozessor gewisse Schwierigkeiten mit der Multiplikation und Division von 32-Bit-Zahlen. Deshalb benutzt MaxonC++ dafür normalerweise korrekt rechnende Bibliotheksfunktionen, die aber recht langsam sind.

Wenn Sie das Flag `"-gm"` setzen, benutzt *MCPPC* trotzdem die normalem Rechenoperationen des Prozessors. Für den korrekten Default steht der Schalter `"-gM"`. Falls Sie aber Code für "68020-oder-höher"-Prozessoren erzeugen, ist diese Einstellung ohnehin obsolet.

5.3.5.5 Optimierungen: -go und -gO

Die zusätzlichen Optimierungen (2.5.2.4.8) werden mit der Option `"-go"` aus- und mit `"-gO"` eingeschaltet. Defaultmäßig sind sie aus. `"-O"` kann als Abkürzung für `"-go"` benutzt werden.

5.3.5.6 Code für besondere Prozessoren: -g00, -g20, -g80 und -g81

Auch wenn sich immer weniger Amiga-User mit einem schlichten 68000er-Prozessor begnügen, erzeugt MaxonC++ normalerweise Code für diesen Prozessor, der dann natürlich auch zu allen "besseren" Prozessoren kompatibel ist. Allerdings haben die Motorolas von 68020 aufwärts einige neue Befehle, mit denen Programme teilweise etwas beschleunigt werden, z. B. bei der Multiplikation und beim Arrayzugriff. Die folgenden Optionen schalten die Codeerzeugung auf einen bestimmten Prozessor um:

`-g00`	68000	(Default)
`-g10`	68010	(identisch mit -g00)
`-g20`	68020	
`-g30`	68030	
`-g40`	68040	

In der vorliegenden Version sind die Optionen `"-g20"`, `"-g30"` und `"-g40"` noch identisch, d. h. es kann nur Code für "ab 68020" und kein spezieller für 68040 erzeugt werden. Auch wenn das in späteren Compilerversionen sicher noch verbessert wird, sollten Sie sich von der CPU-Wahl keine Wunder erwarten: Wenn man sich nicht gerade ein Benchmark maßschneidert, bringen die Befehlssatzerweiterungen der 680x0-Familie nicht übermäßig viel (siehe auch Abschnitt 2.5.2.4.9).

Dagegen bringt die Benutzung der Floating Point Unit (FPU) bei Programmen, die Fließkomma-Operationen benutzen (also die Datentypen "float" und "double"), durchaus einen meßbaren Gewinn. Mit den folgenden Optionen erzeugt MCPPC Code mit oder ohne FPU:

`-g80`	ohne FPU (Default)
`-g81`	68881
`-g82`	68882 (z. Zt. identisch mit -g81)

In der Grundeinstellung, d. h. ganz ohne FPU, benutzt MaxonC++ die Fließkomma-Bibliotheken des AmigaOS, welche bei Vorhandensein einer FPU diese automatisch benutzen. Deshalb bringt auch die FPU-Benutzung letztlich nicht so viel, wie der unbedarfte Benutzer vielleicht erwarten würde.

5.3.5.6 Linken ohne Startup-Code: -gs, -gsi und -gsw

Wie in Abschnitt 4.6 beschrieben, läßt `"-gs"` beim Linken den Startup-Code weg, generiert dafür aber zwei explizit aufrufbare Funktionen `"_InitModules"` und `"_CleanupModules"`. Bei `"-gsi"` werden diese nur erzeugt, wenn sie irgendwo im Projekt referiert werden, und `"-gsw"` gibt zusätzlich eine Warnung aus, wenn eine der beiden Funktionen vorhanden ist, aber trotzdem nirgendwo aufgerufen wird.

5.3.5.7 Code für Link-Libraries: -gl, -gL und -L

Wie in 2.5.3.2 beschrieben, kann MaxonC++ auf Wunsch jede Funktion und jede globale Variable in separaten Hunks ablegen. Da der Linker unreferenzierte Hunks nicht in die ausführbare Programmdatei aufnimmt, eignet sich dieser Modus besonders für einlinkbare Bibliotheken. Mit `"-gL"` oder kurz `"-L"` wird diese Option eingeschaltet. `"-gl"` schaltet wieder in den normalen Modus zurück.

5.3.6 Debugger

5.3.6.1 Maxon Debugger: -b, -bi und -b0

Wenn Sie den Maxon Debugger besitzen, können Sie mit der Option `"-b"` die notwendigen Debugger-Dateien ("*.mdbi*" und "*.mdbg*") erzeugen lassen. Die Option `"-bi"` schreibt zusätzlich ein Icon für die "*.mdbg*"-Datei. `"-b0"` schaltet wieder in den Urzustand.

5.3.6.2 Symbolischer Debugger: -bs und -bs0

Die Option `"-bs"` erzeugt Symboltabellen-Hunks für symbolische Debugger (siehe 2.5.2.4.5).

5.3.7 Warnungen und Fehler

5.3.7.1 Format der Fehlerausgabe: -e, -el und -ei

MCPPC kann Fehlermeldungen auf drei verschiedene Arten ausgeben, nämlich interaktiv, in Kurz- und in Langform. Voreingestellt ist die interaktive Fehlerausgabe, für die auch die Option `"-ei"` steht. Dabei werden zu jeder Fehlermeldung ein kurzer Quelltextausschnitt (ca. 5 Zeilen), die Quelltextposition und eine Eingabeaufforderung ("Weiter mit <return>, Abbbrechen mit <A>:") ausgegeben. Die genaue Fehlerstelle ist im Quelltextausschnitt farbig hervorgehoben. Nach Drücken der <Return>-Taste oder alternativ der <Leertaste> oder <C> wird die Übersetzung fortgesetzt, bei Eingabe <A> wird abgebrochen. Alternativ können Sie auch mit der <ESC>-Taste oder <Ctrl>+<C> abbrechen.

Die Compileroption `"-e"` schaltet auf die Kurzausgabe. Hier werden bei jedem Fehler und jeder Warning nur zwei Zeilen mit der Quelltextposition und der eigentlichen Meldung ausgegeben. Spätestens nach zehn Fehlern wird der Übersetzungsvorgang abgebrochen. Für diese Anzahl können Sie einen beliebigen anderen Wert wählen, indem Sie die Zahl direkt an die Option anhängen, also z. B. `"-e20"` für maximal 20 Fehlermeldungen.

Das Langformat ist eine Mischung aus interaktivem Modus und Kurzform. Dabei werden Quelltext und Fehlermeldung wie bei `"-ei"` ausgegeben, ohne daß auf eine Eingabe gewartet würde. Nach maximal zehn Fehlern macht der Compiler Schluß. Für diese Form der Fehlerausgabe wählen Sie die Option `"-el"`. Auch hier kann wie bei `"-e"` auf Wunsch die maximale Fehleranzahl angehängt werden, z. B. wäre `"-el5"` eine sinnvolle Einstellung, da zehn Meldungen im Langformat in kein halbwegs normales Shell-Fenster passen würden.

5.3.7.2 Farben und Stile: -eb, -eB und -ec

Die doch etwas dröge Textausgabe können Sie optisch ein wenig aufpeppen und dabei zugleich übersichtlicher gestalten. Zum einen haben sie die Möglichkeit, die eigentlichen Fehlermeldungen mit dem Schalter `"-eB"` fett ("Bold") ausgeben zu lassen. Falls das Ihnen doch nicht so gut gefallen sollte, können Sie mit "-eb" wieder auf die Grundeinstellung zurückschalten.

Die Option `"-ecN"` bringt Farbe ins Spiel, zumindest dann, wenn Sie für `"N"` eine Ziffer einsetzen. Die Bildschirmfarbe Nummer `"N"` wird dann nämlich benutzt, um den Quelltextausschnitt von den sonstigen Ausgaben abzuheben. `"-ec1"` entspricht der Standard-Textfarbe und `"-ec0"` theoretisch der Hintergrundfarbe. Da man im letzteren Fall aber nichts lesen könnte, gilt hier die Sonderregel, daß `"-ec0"` den Quelltext in Farbe 2 ausgibt, dabei aber die Fehlerposition in einer anderen Farbkombination als `"-ec2"` hervorhebt.

5.3.7.3 Fehlerdatei: -ef

Zur Einbindung von *MCPPC* in fremde Umgebungen kann es nützlich sein, alle Fehlermeldungen in eine Datei umzuleiten. Das ermöglicht eine Angabe der Art `"-ef <name>"`. Dadurch werden alle Fehlermeldungen und Quelltextpositionen in die angegebene Datei geschrieben. Der Unterschied zu einer schlichten Ausgabeumleitung besteht darin, daß erstens `"-ef"` zusätzlich zur normalen Bildschirmausgabe arbeitet und zweitens alle Statusmeldungen nicht in der Datei erscheinen.

5.3.7.4 Optionale Warnungen: -w

Die unter 2.5.2.5 beschriebenen zusätzlichen Warnungen gibt es natürlich auch beim CPPC. Man schaltet eine Warnung ein, indem man ihren Kennbuchstaben GROß hinter das `"-w"` setzt, während ein kleiner Buchstabe eine Warnung ausschaltet.

Dies sind die Warnungen und ihre Kennbuchstaben:

- `a` (Assignment): Der Operator "=" taucht in einer Bedingung auf - vielleicht der bekannte `"=="`-Anfänger-Fehler? Sie können eine solche Warnung übrigens durch eine zusätzliche Klammerung unterdrücken, indem Sie z. B. `"if ((a=b))"` statt `"if (a=b)"` schreiben.
- `c` (Conversion): Unsichere Typkonvertierung ohne Cast, z. B. `double -> int`
- `e` (Empty Statement): Anweisung hat keinen Code, z. B. `"42;"`
- `f` (Function prototypes): Funktion ohne Prototyp aufgerufen (nur in C sinnvoll, da in C++ sowieso ein Error)
- `n` (Nested comments): Ein `"/*"` taucht in einem Kommentar auf.
- `o` (Old-Style): Über Parameterlisten im alten "K&R"-Stil meckern (nur in C sinnvoll, da in C++ sowieso ein Error)
- `p` (Pragma): Unbekanntes `"#pragma"`
- `r` (Return): Funktion mit Rückgabewert hat keine `"return"`-Anweisung
- `t` (Temporary): Temporäres Objekt wurde eingeführt, um ein nicht-`"const"`-Referenz zu initialisieren - was in C++ seit dem 2.0-Standard eigentlich sogar ein Error ist.

`v` (Variable): Lokale Variable wird nie benutzt, oder benutzt, aber nie initialisiert.

`m` (Macro): Ein Argument für ein Preprozessor-Makro erstreckt sich über mehr als 8 Zeilen oder 200 Token. Es liegt der Verdacht nahe, daß hier schlicht eine schließende Klammer vergessen wurde.

Defaultmäßig sind `"-wEPRT"` aktiviert. Um z. B. Warnungen bei fehlenden Prototypen ein- und solche bei unbekannten `"#pragma"` auszuschalten, setzen Sie einfach die Option `"-wFp"` oder alternativ die beiden Optionen `"-wF -wp"`.

Um alle Warnungen einzuschalten - übrigens ein Zeichen für guten Programmierstil - ist demzufolge etwas Tipparbeit vonnöten. Um hier die Hemmschwelle möglichst niedrig zu halten, gibt es eine Abkürzung: Ein `"+"` direkt hinter dem `"-w"` schaltet alle Warnings ein. Bitte beachten Sie, daß demzufolge `"-w+v"` NICHT bedeutet, daß die Warning `"v"` zusätzlich zu denbisherigen eingeschaltet wird, sondern daß ganz im Gegenteil zuerst alle Warnings ein- und anschließend `"v"` wieder ausgeschaltet wird.

Umgekehrt können Sie auch mit `"-w-"` alle Warnungen ausschalten, wovor ich Sie aber dringendst warnen möchte. Ein sauberes Programm sollte zu überhaupt keinen Warnungen Anlaß bieten, allenfalls die `"a"`-Warnung ist eventuell als Ausdruck Ihres persönlichen Programmierstils vertretbar. Wenn Sie Warnungen löblicherweise tatsächlich so ernst nehmen, dürfte auch der folgende Abschnitt für Sie interessant sein.

5.3.7.5 Warnungen wie Fehler behandeln: -ew und -eW

Durch die Option `"-ew"` werden Warnungen wie Fehlermeldungen gewertet. `"-eW"` schaltet wieder zurück in den normalen Modus.

5.3.8 Linker

5.3.8.1 Library-Pfade: -j

Der Pfad, über den der Linker die Bibliotheken lädt, ist mit "*MCPP:lib*" voreingestellt und kann mit der Option `"-j"` verändert werden, z. B. `"-j dh1:cpp/lib"`. Genau wie bei der Option `"-i"` (5.2.3.2) kann auch hier ein mit `"|"` getrennter Doppelpfad wie z. B "*ram:clib|MCPP:lib*" angegeben werden, um die benutzten Dateien automatisch auf ein schnelleres Speichermedium umkopieren zu können. Im Gegensatz zu den Include-Pfaden kann für Libraries aber immer nur ein Pfad bzw. nur ein Doppelpfad definiert werden.

5.3.8.2 Linker-Libraries: -l

Um Linker-Bibliotheken (2.5.3.3) einbinden zu lassen, setzt man ihren Namen einfach hinter die Option `"-l"`, z. B. wird bei

```
mcppc ami.c -lamiga
```

die "*amiga.lib*" eingebunden. Hinter dem `"-l"` darf optional auch ein Leerzeichen stehen, und natürlich kann man auch mehrere Libraries einbinden.

5.3.9 Die integrierte Projektverwaltung: -m, -mx und -mX

Das Make-Utility gibt es nicht nur im der integrierten Umgebung "*MCPP*", sondern auch im *mcppc*. Man übersetzt anhand eines Makefiles, indem man die Option `"-m"` benutzt, z. B.

```
mcppc -m makefile
```

Die Make-Datei muß die in Abschnitt 3.3 beschriebene Form besitzen. Man kann zusätzliche globale Optionen für die Übersetzung angeben, z. B.

```
mcppc -m makefile -w200 -gO -gd
```

Mit diesen Einstellungen werden dann die C++-Objektdateien übersetzt, sofern in deren Compileranweisung nicht expilzit gegenteilige Optionen stehen. Auch Linkeroptionen können hier angegeben werden.

Namen zusätzlicher Text- und/oder Objektdateien werden im Make-Modus in der Kommandozeile naheliegenderweise nicht akzeptiert.

Die Pflege einer Projektdatei ist eine mühsame Angelegenheit. Wenn Sie keine Lust haben, die Abhängigkeiten von Hand zu setzen, oder fürchten, dabei etwas falsch zu machen, können Sie (zusätzlich zu `"-m"`) die Option `"-mX"` benutzen. Dann paßt *MCPPC* beim Compilieren auf, welche Quelltextdateien für welche Objektdatei benötigt werden, und aktualisiert anschließend das Makefile.

Aber Vorsicht: Ihre womöglich kunstvoll und elegant geschriebene Projektdatei wird dabei durch ein schnödes MCPP-Standard-Makefile geplättet. Insbesondere verschwinden dabei Kommentare, Makros und Regeln auf nimmerwiedersehen.

Wie Sie vielleicht schon vermutet haben, ist `"-mx"` der zu `"-mX"` passende Ausschalter.

Zum Schluß noch ein Tip: Sie können sich auch ein ganz neues Makefile erzeugen lassen, wenn Sie `"-mX"` ohne `"-m"` verwenden. Bei einem Aufruf wiem

```
mcppc x.c y.c z.c -mX
```

erzeugt *MCPPC* eine Projektdatei namens "*makefile*", in der die drei Objekte "*x.o*", "*y.o*" und "*z.o*" mit allen Compileranweisungen, Quelltexten und Anhängigkeiten stehen.

5.3.10 Arbeitsspeicher: -w

MaxonC++ alloziert zwar den Speicherplatz für die meisten internen Angelegeheiten automatisch und dynamisch, aber bei der Codegenerierung wird trotzdem ein zusammenhängender Speicherbereich benötigt, dessen Größe mit 80 KByte voreingestellt ist. Sollte bei einem umfangreichen Programm einmal eine Meldung wie **Workspace overflow!** ausgegeben werden, sollten Sie die Größe dieses Speicherbereichs mit der Option `"-w"` behutsam erhöhen, z. B. `"-w150"` für 150 KByte.

5.3.11 Vorcompilierte Headerdateien: -H, -h und -h0

Seit Version 3 kann MaxonC++ Programme vorcompilieren, d. h. an einer vom Benutzer frei wählbaren Stelle kann der Compiler alle seine internen Daten - und damit die bisher gelesenen Deklarationen und Definitionen - abspeichern. Beim nächsten Compilerdurchlauf kann diese Datei dann sehr schnell geladen und so viel Zeit gespart werden.

Dieses Feature ist dafür gedacht, die Programmentwicklung zu beschleunigen, wenn sehr viele Header-Dateien benutzt werden, z. B. die AmigaOS-Includes oder die MaxonC++ Klassenbibliothek "*EasyObjects*". Damit der Compiler aber weiß, wo die Header aufhören und Ihr eigenes Programm anfängt, muß man ihm diese Stelle mit einer Zeile **"#pragma header"** markieren.

Am besten schreiben Sie in den Quelltext jeder Übersetzungseinheit zunächst die **"#include's"** all jender Headerdateien, die Sie nicht selbst geschrieben haben oder an denen sich aus anderen Gründen nichts mehr ändern wird. Dann folgt die Zeile **"#pragma header"**, und anschließend folgen die Include-Anweisungen für Ihre eigenen Header-Dateien und der Rest Ihres Programms.

Ohne besondere Optionen, oder mit dem expliziten Ausschalter **"-h0"**, hat **"#pragma header"** keinerlei Wirkung. Wenn Sie aber eine Option **"-H <name>"** angeben, hält der Compiler, sobald das Pragma erreicht wird, kurz inne und schreibt alle Definitonen und Deklarationen in die Datei **"<name>"**. Danach arbeitet er weiter, als sei nichts geschehen.

Um das Programm anschließend beschleunigt erneut zu compilieren, wählen Sie eine Option wie **"-h <name>"**. Dann liest der Compiler die vorcompilierten Header aus der angegebenen Datei, sucht im Quelltext nach dem **"#pragma header"** und setzt dort mit seiner Übersetzungstätigkeit ein.

Da alles, was vor dem **"#pragma header"** steht, komplett ignoriert wird, muß es folglich in der Haupt-Quelltextdatei der Übersetzungseinheit - und nicht etwa in einer Includedatei - stehen.

Denken Sie bitte daran, daß *MCPPC* nicht prüft, ob der Inhalt der vorcompilierten Header-Dateien überhaupt noch mit dem ursprünglichen Quelltext übereinstimmt. Deshalb empfehle ich noch einmal dringend, nur solche Header-Dateien vorzucompilieren, an denen sich während der Programmentwicklung nichts mehr ändert.

5.4 Voreinstellungen mit "CPPOPT"

Wenn es Ihnen zu mühsam ist, bei jedem Compileraufruf Ihre ieblings-Optionen von Hand einzutippen, können Sie sich natürlich in einer geeigneten Shell ein passendes Alias definieren, z. B. haben Sie sich durch

```
alias cc2 "mcppc -pc -w300 "
```

einen C-Compiler mit großem Workspace geschaffen. Eleganter geht dies aber mit der Environment-Variablen **"CPPOPT"**, die freilich erst ab Betriebssystemversion 2.0 unterstützt wird:

```
setenv CPPOPT -pc -w300
```

stellt die beiden genanntem Optionen als Default ein. Die Variable wird so behandelt, als würde ihr Wert an den Anfang der Kommandozeilenargumente gestellt. Deshalb können Sie hier theoretisch nicht nur irgendwelche Optionen wählen, sondern z. B. auch mit

```
setenv CPPOPT test.c
```

zusätzliche Quelltexte oder Objektdateien angeben. Dies dürfte aber nicht unbedingt sinnvoll sein.

5.5 Zusammenfassung

Es folgt noch einmal eine kurze, alphabetisch geordnete Übersicht über alle CPPC-Optionen:

`-a`	erzeugt für jedes übersetzte Textfile einen Assemblerquelltext.
`-as`	C-Quelltext wird in Assembler-Source als Kommentar eingefügt.
`-a0`	Keine Assembler-Ausgabe (Default)
`-b`	Debug-Info-Datei erzeugen.
`-bi`	Wie `"-b"`, aber Icon für "*.mdbg*"-File erzeugen.
`-b0`	Ohne Debug-Info-Datei (Default)
`-bs`	"Symbol-Hunk" für symbolischen Debugger erzeugen.
`-bs0`	Kein Symbolhunk (Default)
`-c`	Die Quelltexte werden compiliert und die entsprechenden ".o"-Dateien erzeugt. Das Programm wird nicht gelinkt.
`-cx`	Wie `"-c"`, aber das ganze Projekt wird trotzdem gelinkt und eine ausführbare Programmdatei erzeugt.
`-cX`	Übersetzen und linken, aber keine Objektdateien schreiben (Default)
`-d NAME`	definiert ein Preprozessor-Symbol. Optional kann unmittelbar hinter den Namen ein `"="` und der gewünschte Wert (nur ein Token) gesetzt werden, andernfalls ist das Symbol als „leer" definiert.
`-eB`	Fehlermeldungen fett schreiben
`-eb`	Fehlermeldungen normal schreiben (Default)
`-ecN`	Bei Fehlermeldungen Quelltext in Farbe Nummer `"N"` ausgeben
`-ef xxx`	Fehlermeldungen in Datei "*xxx*" schreiben
`-ei`	Bei jeder Compiler-Fehlermeldung wird der Benutzer gefragt, ob abgebrochen werden soll (Default)
`-e[N]`	Fehlermeldungen werden ausgegeben, ohne daß Interaktion des Benutzers verlangt wird. Spätestens nach dem `"N"`-ten Fehler wird abgebrochen (Default ist 10).
`-el[N]`	Wie `"-e"`, aber zusätzlich mit Quelltextausgabe
`-ew`	Warnungen wie Fehler behandeln
`-eW`	Warnungen ausgeben und ansonsten ignorieren (Default)
`-gb`	Breakpoints setzen
`-gb0`	Keine Breakpoints setzen (Default)
`-gd`	Datenmodell "Small Data".

`-gD` Datenmodell "Large Data" (Default)

`-gh xxx` legt die maximale Größe (in KByte) fest, bis zu der gleichartige Hunks vom Linker zusammengefaßt werden. Default ist 50.

`-gl` Alle Funktionen in einer Section (Default)

`-gL` Library-Modus: Für alle Funktionen und jede globalen Variablen eigene Sections erzeugen

`-gm` MULS/DIVS-Operationen des Prozessors benutzen (Default: Bibliotheksfunktionen für korrekte 32-Bit-Arithmetik).

`-go` ohne Optimierungen (Default)

`-gO` mit Optimierungen

`-gs` Ohne Startup-Code linken

`-gsi` Wie `"-gs"`, aber Hunk mit `"InitModules"` und `"CleanupModules"` nur erzeugen, wenn diese Symbole auch irgendwo benutzt werden (praktisch, wenn man *MCPPC* als schlichten Linker z. B. für Assembler-Projekte mißbraucht).

`-gsw` Wie `"-gsi"`, aber zusätzlich Warnung ausgeben, wenn `"InitModules"` oder `"CleanupModules"` zwar nicht leer sind, aber trotzdem nirgendwo benutzt werden. Dann hat der Programmierer diese Aufrufe offenbar vergessen.

`-g00` Code für 68000 erzeugen (Default)

`-g20` Code für 68020 erzeugen

`-g30` Code für 68030 erzeugen

`-g40` Code für 68040 erzeugen

`-g80` Code ohne FPU-Benutzung erzeugen (Default)

`-g81` Code für 68881-FPU erzeugen

`-g82` Code für 68882-FPU erzeugen

`-H xxx` vorcompilierte Header-Dateien abspeichern

`-h xxx` vorcompilierte Header-Dateien laden

`-h0` Ohne vorcompilierte Header übersetzen (Default)

`-i xxx` Den Pfad bzw. die Pfade, über den *MCPPC* die Includefiles sucht, setzen. Mehrfaches `"-i"` für mehrere Pfade ist möglich. Defauli ist "*MCPP:include*". Tritt in einem Dateinamen ein `"|"` auf, werden die gelesenen Files automatisch vom zweiten in den ersten Pfad umkopiert und dabei kompaktiert, z. B. "*-i ram:include|c++:include*"

`-j xxx` definiert analog zu `"-i"` den Pfad bzw. den Doppelpfad, über den die Libraries gesucht werden. Hier kann aber nur ein Pfad bzw. nur ein Doppelpfad (mit `"|"`) angegeben werden. Default ist "*MCPP:lib*".

`-l xxx` Die Library "*xxx.lib*" wird aus dem Libs-Verzeichnis eingebunden.

`-L` Abkürzung für `"-gL"`

`-m xxx` übersetzt nach Makefile "*xxx*".

`-mX` Abhängigkeiten in Makefile automatisch setzen

`-mx` Makefile bloß lesen (Default)

`-o xxx` legt einen anderen Namen für die Exe- bzw. die Objektdatei fest.

`-O` Abkürzung für `"-gO"`

`-pc` schaltet in ANSI C-Modus
`-pp` C++-Modus (Default)
`-px` Ohne Exception Handling
`-pX` Exception Handling ermöglichen
`-t` Nur fehlerfreie Template-Funktionen erzeugen
`-T` Alle benötigten Template-Funktionen erzeugen
`-w xxx` Warnings ausgeben. Es kann eine beliebige Folge von Groß- und Kleinbuchstaben folgen, wobei ein GROSSER Buchstabe die Warnung einschaltet.

Folgende Warnungen gibt es:

`a` (Assignment): Operator `"="` taucht in Bedingung auf - vielleicht der bekannte `"=="`-Anfänger-Fehler?
`c` (Conversion): Unsichere Typkonvertierung ohne Cast, z. B. `double -> int`
`e` (Empty Statement): Anweisung hat keinen Code, z. B. `"42;"`
`f` (Function prototypes): Funktion ohne Prototyp aufgerufen (nur in C sinnvoll, da in C++ sowieso ein Error)
`n` (Nested comments): `"/*"` in Kommentar
`m` (Macro): Ein Argument für ein Preprozessor-Makro erstreckt sich über mehr als 8 Zeilen oder 200 Token. Es liegt der Verdacht nahe, daß hier schlicht eine schließende Klammer vergessen wurde.
`o` (Old-Style): Über Parameterlisten im alten "K&R"-Stil meckern (nur in C sinnvoll, da in C++ sowieso ein Error)
`p` (Pragma): Unbekanntes `"#pragma"`
`r` (Return): Funktion mit Rückgabewert hat kein `"return"`.
`t` (Temporary): Temporäres Objek wurde eingeführt, um ein nicht-`"const"`-Referenz zu initialisieren - was seit dem 2.0-Standard eigentlich sogar ein Error ist.
`v` (Variable): Lokale Variable wird nie benutzt bzw. benutzt, aber nie initialisiert.

Defaultmäßig sind `"-wEPRT"` aktiviert.

`-wN` Setzt den compilerinternen Workspace auf "N" KByte. Default sind 80 KByte, und das sollte reichen.
`-w+` Alle Warnungen ausgeben
`-w` Keine Warnungen ausgeben
`-X` Abkürzung für `"-pX"`

Anhang A

Veränderungen seit der Version 1.0

Seit der Version 1.0 hat sich bei Maxon C++ offensichtlich einiges getan. Die diversen Bug-Fixes möchten wir hier nicht aufführen, denn meist ist die exakte Beschreibung der Fehler dermaßen kompliziert, daß sich kaum jemand das alles durchlesen würde. Auch viele Details in der Code-Erzeugung wurden verbessert, was der Programmierer aber nicht unbedingt wissen muß, denn das äußert sich ausschließlich in teilweise kompakteren und schnelleren Programmen.

Die "großen" Erweiterungen und Veränderungen, etwa die Exceptions und Templates oder die völlig überarbeitete Entwicklungsumgebung, dürften Ihnen schon aus dem Vorwort des Benutzerhandbuchs oder durch bloßen Augenschein bekannt sein. Deshalb beschränken wir uns hier auf einen Hinweis auf Abschnitt 5.1 des Benutzerhandbuchs, wo die Änderungen an der CLI-Compilerversion "*mcppc*" dargestellt werden, und gehen gleich zu den vielen kleinen, aber feinen Änderungen über:

A-1. Compiler

A-1.1 Preprozessor

- **"Pseudo-Vorcompilieren" von Includes:**
  Beim Umkopieren von Includes werden diese vorgescannt - bringt ein bißchen 'was an Compilezeit und spart enorm Speicher, wenn in die RAM-Disk umkopiert wird.
- Die interne Compiler-Hashtable ist größer und kostet trotzdem weniger Speicher.
- Makro `"__DATE2__"` liefert das Datum in vernünftigem Format.
- Makro `"__MAXON__"` wurde auf `"30"` geändert.
- Das `"#pragma header"` wurde zwecks Vorcompilieren von Header-Dateien eingeführt.

A-1.2 Parser und Sprachstandard

- Pointer auf Member-Funktionen sind jetzt implementiert.
- Beim bedingten Ausdruck (`c ? a : b`) mit konstanter Bedingung `"c"` wird die Fallunterscheidung wegoptimiert, und der ganze Ausdruck ist dann u. U. konstant (letzteres ist strenggenommen ein Bug-Fix)
- Bei der Pointer-Subtraktion kann jetzt die Differenz zwischen einen `"T *"` und einen `"const T *"` berechnet werden. Was hier der Standard vorschreibt, ist nicht ganz klar, aber die neue Regelung ist sinnvoll und harmlos.
- Operatoren `"+="`, `"|="` etc. funktionieren jetzt auch auf Bitfeldern.

- Neue Fehlermeldung, wenn im Small Data Model ein Parameter in Register A4 übergeben wird, und in Large Data ist das jetzt fehlerfrei möglich.
- Warnung `#10` für „Verdächtiges" `'='` in Bedingung wird nicht mehr ausgegeben, wenn die Zuweisung in Klammern steht (z. B. `"if ((a=b))..."`)
- In `"int main()"` wird bei fehlendem `"return"` nicht mehr gewarnt.
- Sinnreichere Fehlermeldung `"Class/struct xxx undefined"` bei Memberzugriff auf undefinierte Struktur
- Bei abstrakten Klassen wird ggf. der Name der fehlerhaften Basisklasse in der Fehlermeldung ausgegeben (Fehler `#179` und `#181`).
- Die Schlüsselworte `"catch"`, `"throw"` und `"try"` sind gesperrt, wenn das Exception-Handling nicht per Compiler-Option eingeschaltet wurde.

A-1.3 Codeerzeugung

- Datentyp `"float"` wurde von "FFP" auf "IEEE-Single" umgestellt.
- Im "Library-Modus" werden alle Funktionen und globale Variablen in separate Hunks gepackt.
- Alignment bei mehrdimensionalen Vektoren ist besser gelöst, z. B. ist `"char x[10][9]"` jetzt 90 (bisher: 100) Bytes groß.
- 68020/30-Code mit Option `"-g20"` bzw. `"-g30"` (benutzt vorerst nur Langwort- Multiplikation und -Division sowie Skalierung bei Index-Adressierung)
- Direkte FPU-Unterstützung
- Bedingungstest bei `"while"`-Schleifen steht jetzt am Schleifenende (spart einen Branch pro Durchlauf).
- Inlining von Funktionen geschieht auch, wenn Funktionsdefinition hinter Funktionsaufruf steht
- Identische Zeichenketten werden nur noch einmal abgespeichert
- Längst überfällige Codeverbesserungen, z. B. wird `'*p++'` ggf. in `'(aN)+'` übersetzt

A-1.4 Linker

- Alle Varianten von `"main"` werden ins "*.mdbg*"-File eingetragen (`"main_"`, `"main_iPPc"`, `"_main"` und `"_wbmain"`).
- Bei Linkerfehlern werden die Linkernamen nach Möglichkeit in C++-Funktionsdeklarationen rückübersetzt

☞ Jetzt sind bis zu 1000 Hunks pro geladener Objektdatei möglich (früher nur 100)

A-2. Bibliotheken

☞ Stubs für ASL-, DOS- und Intuition-Tag-Funktionen (erspart amiga.lib)

☞ Die Funktion **"wbparse"** erleichtert den Workbench-Startup, insbesondere im Zusammenhang mit dem Include *<wbstartup.h>*.

☞ Die *"ctype.h"*-Funktionen sind um eine Winzigkeit schneller und kompakter geworden.

☞ Die Cleanup-Priorität 0 (**"_EXIT_0_..."**) wird nur noch fürs Beantworten der Startup-Message (bei Workbench-Startup) benutzt. Wer Assemblermodule schreibt, die eine solche Cleanup-Funktion besitzen, sollte die Priorität 0 deshalb nicht mehr benutzen und bedenken, daß alle anderen Library-Module jetzt ihr Resource Tracking o. Ä. auf Level 1 und 2 durchführen.

☞ Die Funktionen **"memmove"** und **"memcpy"** sind jetzt identisch, kommen also beide mit überlappenden Speicherbereichen klar. Außerdem wurden sie durch langwortweises Kopieren (falls möglich) beschleunigt.

☞ **"realloc"** arbeitet jetzt Amiga-konform (siehe auch Abschnitt 1.1.4 im Referenzhandbuch)

A-3. Sonstiges

☞ Defaults für diverse Pfade und Dateinamen wurden vereinheitlicht (z. B. logisches Verzeichnis *"MCPP:"* und alle Executables in *"MCPP:bin/"*)

☞ Auf vielfältigen Wunsch setzt **"-o"** in Verbindung mit **"-c"** jetzt auch den Namen der Objekt-Datei. Werden mehrere Objektdateien auf einmal übersetzt, bezieht **"-o"** sich ausschließlich auf den erstgenannten Quelltext.

☞ Der Chunk **"HUNK"** wird in den *".mdbi"*-Dateien nicht mehr geschrieben (war überflüssig und wäre durch die massig vielen möglichen Hunks etwas aufwendig gewesen)

☞ *MCPP* definiert für Edward jetzt das Symbol **"MCPP3"** (wg. *DEF-Datei*).

Anhang B – Compiler Fehlermeldungen

MaxonC++ kennt etwa 250 Fehlermeldungen und ein gutes Dutzend Warnungen. In der vorliegenden Version sind diese noch ausschließlich Englisch, für die nähere Zukunft ist aber vorgesehen, sie zu lokalisieren. Deshalb werden sie hier nach Nummern sortiert. Beachten Sie bitte, daß Fehler und Warnungen getrennt numeriert sind.

B-1. Warnungen

MaxonC++ bietet dreizehn Warnungen, die größtenteils einzeln an- und abschaltbar sind:

1: Nested comment

Die Zeichenfolge `"/*"` innerhalb eines Kommentars ist verdächtig, da hier möglicherweise schlicht vergessen wurde, einen vorhergehenden Kommentar mit `"*/"` zu beenden.

2: No prototype for function "<name>".

Eine Funktion ohne Prototyp wurde aufgerufen. In ANSI C ist es zwar strenggenommen nicht erlaubt, Funktionen ohne Prototyp aufzurufen, aber aus Gründen der Kompatiblität mit dem hoffnungslos veralteten und ebenso hoffnungslos unausrottbaren "K&R"-Standard gibt es dafür im C-Modus nur eine abschaltbare Warnung. In C++ werden Sie diese Warnung nicht erhalten, denn da ist das ein echter Fehler.

3: Function "<name>" has no "return" statement.

Wenn in einer Funktion, deren Ergebnistyp nicht `"void"` ist, keinerlei `"return"`-Anweisung auftaucht, wurde diese offensichtlich vergessen.

4: Statement has no effect at all.

Es wurde eine Anweisung wie `"42;"` gefunden, die zwar syntaktisch absolut korrekt, aber offensichtlich sinnlos und deshalb wahrscheinlich nicht so beabsichtigt ist. Besonders häufig dürfte diese Warnung wohl beim Aufruf einer parameterlosen Funktion ohne Argumentliste (z. B. `"f;"`) auftreten.

5: Variable <xxx> declared but never used.

Natürlich dürfen Sie so viele Variablen deklarieren, wie Sie wollen, und niemand zwingt Sie, diese auch zu benutzen. Eine entsprechende Warnung, z. B. bei einer Variablen, die einmal benutzt worden war, inzwischen aber redundant ist und nur Speicherplatz kostet, ist aber sicher bisweilen ganz angenehm.

Es wird hier aber ausschließlich bei lokalen Variablen von Funktionen gewarnt, da der Compiler nicht übersehen kann, ob eine globale statische Variable vielleicht in einer anderen Übersetzungseinheit benutzt wird. Funktionsparameter werden in dieser Überprüfung ebenfalls nicht berücksichtigt.

6: This seems to be an old-style parameter list.

Der Compiler hat eine Parameterliste im alten Kernighan-Ritchie-Stil gefunden, oder glaubt es zumindest (es könnte auch ein beliebiger anderer Syntaxfehler sein). Diese Parameterlisten sind nicht schön, weshalb der Compiler bei Bedarf mittels einer Warnung seine Verachtung darüber zum Ausdruck bringen kann.

7: Unknown #pragma (maybe misspelled?)

Ein unbekanntes Pragma wurde gefunden - möglicherweise verdächtig.

8: Temporary object for non-const reference introduced.

Eine Referenz auf einen nicht konstanten Datentyp wurde initialisiert, wobei aber ein temporäres Objekt eingeführt werden mußte. Dies ist seit dem C++ 2.0-Standard strengenommen nicht erlaubt und außerdem immer eine suspekte Konstruktion.

9: Unsafe type conversion

In der vorliegenden Compiler-Version wird diese Warnung nur ausgegeben, wenn ein Fließkommawert ohne explizite Typkonvertierung (also ohne Cast) in einen ganzzahligen Typen umgewandelt wurde. Weitere "unsichere" Umwandlungen werden mit dieser Warnung eventuell in spätern Compilerversionen gemeldet.

10:Suspect "=" in condition.

Eine Wertzuweisung wurde als logische Bedingung benutzt, z. B. wie in `"if (a=b)"`. Das ist zwar allgemein beliebt, aber möglicherweise auch ein Versehen.

Wenn solche Konstrukte zu Ihrem persönlichen Programmierstil gehören, sie aber trotzdem deratigen Fehlern vorbeugen möchten, können Sie die Warnung überlisten: Durch eine zusätzliche Klammer, also z. B. `"if ((a=b))"`, an den Stellen, wo Sie es wirklich so meinen, wird die Warnung unterdrückt.

11:Precompiled header file "<name>" not found.

Vorcompilierte Headerdateien können ganz schön groß werden, weshalb man sie beim Aufräumen der Festplatte gerne wegschmeißt. Deshalb macht Maxon C++ keinen großen Aufstand, eine Headerdatei, die gelesen werden soll, einmal nicht vorhanden ist: Die Warnung Nummer 11 wird ausgegeben und das Programm anstandslos übersetzt, nur eben ohne die Header-Datei.

12:Variable <name> never initialized.

Eine lokale Variable wurde deklariert und benutzt, aber nie initialisiert, was offensichtlich ein Fehler ist. Der Compiler ist an dieser Stelle aber ziemlich leicht hinters Licht zu führen: Wenn irgendwo die Adresse der Variablen genommen wird, verfolgt er das nicht mehr weiter und geht einfach davon aus, daß die Variable irgendwie und irgendwann über diesen Zeiger einen Wert erhält. Da bei jedem Vektor-Zugriff implizit der Adressoperator auf den Vektor angewamdt wird, entfällt diese Warnung bei Vektoren. Außerdem ist der Compiler nicht schlau genug, durchgeknallte Anweisungen wie `"i &= 0;"` oder `"j -= j;"` als Initialisierung zu erkennen - aber wer macht auch schon sowas?

13: Very long argument for Macro "<xxx>" - probably missing ")".

Ein Argument für ein Makro erstreckt sich über mahr als 8 Zeilen oder 200 Token. Deshalb liegt der Verdacht nahe, daß hier schlicht eine schließende Klammer vergessen wurde.

B-2. Fehlermeldungen

1: Memory overflow!

Der Speicher ist voll. Die Fehlermeldung bezieht sich NICHT auf den Compiler-Workspace (der kann frühestens während der Codegenerierung überlaufen), sondern auf den freien Systemspeicher. Abhilfe: So viel wie möglich RAM freimachen, oder eine Speichererweiterung kaufen...

2: String constant exceeds line.

Die Zeichenkette geht über das Zeilenende hinaus, oder die abschließenden Anführungszeichen wurden vergessen.

3: Char constant: unmatched '.

Ein einfaches Hochkomma `"'"` leitet eine Zeichenkonstante ein und kann deshalb immer nur paarweise auftreten.

4: Preprocessor command expected.

Hinter einem `"#"` wird eine Preprozessor-Anweisung stehen.

5: Identifier expected.

Hier wird ein Bezeichner erwartet. Im Gegensatz zur wortgleichen Meldung 60 tritt diese im Zusammenhang mit dem Preprozessor auf, z. B. bei einer fehlerhaften Makro-Definition.

6: Macro "<name>" redefined.

Das Preprozessor-Makro `<name>` wurde schon einmal anders definiert.

7: ")" expected.

Ein Preprozessor-Fehler: die abschließende Klammer bei Makrodefinition mit Parameterliste fehlt.

8: "##" at end of Macro.

Der Operator `"##"` verbindet innerhalb von Makrodefinitionen zwei Symbole und kann deshalb niemals am Ende des Makros stehen.

9: Macro expansion: "(" expected.

Das Makro erwartet eine Parameterliste.

10:"," expected.

Hier fehlt ein Komma, oder zumindest glaubt der Compiler das.

12:Compiler is in Panic!

Compiler-intern ist etwas höllisch schiefgelaufen! MaxonC++ ist ziemlich mißtrauisch programmiert und überprüft hin und wieder sich selbst. Wird dabei festgestellt, daß eine Situation vorliegt, die eigentlich unmöglich ist, steigt der Compiler mit dieser Fehlermeldung aus. Es handelt sich hier um einen Bug des Compilers, nicht um einen Fehler Ihres Programms! Wenn dieser Fehler auftritt, melden Sie sich bitte umgehend bei MAXON - am besten wäre, wenn Sie dabei möglichst präzise schildern könnten, wo und unter welchen Umständen der Fehler auftrat.

Als Notbehelf können Sie versuchen, das Programm an der bemängleten Stelle anders zu formulieren. Wahrscheinlich (hoffentlich) gerät der Compiler aber nur in Folge zuvor gemeldeter Fehlern in Panik.

13:Macro expansion: Illegal use of "#" or "##".

Die Token-Operatoren `"#"` und `"##"` werden im vorliegenden Makro falsch angewendet (Tutorial 5.2.2).

14:Recursive definition of macro.

Das Makro taucht direkt oder indirekt in der eigenen Definition auf.

15:Macro name expected.

Hier wird ein gültiger Makroname, also ein Bezeichner, erwartet.

16:"Garbage" at end of preprocessor command.

Die Preprozessoranweisung scheint vollständig zu sein, aber dahinter steht in der gleichen Zeile noch etwas nicht näher Identifizierbares.

17:Filename expected.

Hinter `"#include"` oder `"#line"` wird hier ein gültiger Dateiname erwartet.

18:Can not open file "<name>".

Der Preprozessor findet eine Includedatei nicht. Sie sollten im Zweifelsfall insbesondere überprüfen, ob Sie den Include-Pfad sinnreich definiert haben.

19:Can not copy file "<source>" to "<dest>".

Der Preprozessor hat irgendwelche Probleme, eine Include-Datei umzukopieren. Das kann die unterschiedlichsten, aber meist offensichtlichen Ursachen haben.

20:Overflow at nesting of Includes (max. depth is 25)

Der Preprozessor kann Includedateien höchstens 25-fach ineinander verschachteln (Änderung dieses Limits in weiteren Versionen vorbehalten). Bei einer derartigen Schachtelung liegt ohnehin der Verdacht nahe, daß Sie eine Art endlose Rekursion programmiert haben.

21:Can not open math library.

Für interne Fließkommaberechnungen, z. B. um Ausdrücke wie `"2*3.14159"` schon zur Compile-Zeit auswerten zu können, benötigt der Compiler die "mathieeedoubbas.library".

22:No matching "#if".

Der Preprozessor stieß auf ein `"#endif"` oder `"#else"`, für das es kein `"#if"` gibt.

23:At least one "#endif" is missing.

Am Ende des Quelltexts stellt der Preprozessor fest, daß wenigstens ein `"#if - #endif"`-Preprozessor-Konstrukt noch nicht abgeschlossen ist. Ich wünsche viel Spaß bei der Suche nach dem `"#if"`...

24:Error in condition.

In der Bedingung eines Preprozessor-`"#if"` wurde ein syntaktischer Fehler gefunden.

25:User Error: ...

Diese Fehlermeldungen werden durch die Preprozessor-Anweisung `"#error"` gezielt provoziert.

26:Line number expected.

In der Preprozessoranweisung `"#line"` wird als erster Parameter eine ganzzahlige Zeilennummer erwartet.

27:Pseudo-filename expected.

Als optionaler zweiter Parameter von `"#line"` wird ein Dateiname in Form eines Strings erwartet.

29:Compiler stack overflow!

Der Stack des Compilers ist übergelaufen. Am besten versuchen Sie, *MCPP* bzw. *mcppc* nach Heraufsetzen des Stacks erneut zu starten. Beachten Sie bitte, daß dies nichts(!) mit der Eintragung "Laufzeit-Stack" in der System-Dialogbox von *MCPP* zu tun hat.

Wenn Sie den mcppc benutzen oder die Entwicklungsumgebung *MCPP* von der Shell aus gestartet haben, können Sie mit der Shell-Anweisung "stack" zuerst die aktuell gültige Stackgröße feststellen und dann einen geeigneten höheren Wert setzen. 10000 Bytes sind normalerweise genug, und mit 20000 sind Sie klar auf der sicheren Seite.

Falls Sie aber *MCPP* von der Workbench aus durch Anwählen und Öffnen eines Icons gestartet haben, sollten Sie auf dieses Icon den Workbench-Menüpunkt **"Information"** anwenden. In der Dialogbox, die sich dann öffnet, können Sie die gewünschte Stackgröße eintragen.

30:Too many errors.

Spätestens nach 50 Fehlern weigert sich MaxonC++ weiterzumachen. Warnungen werden dabei normalerweise nicht mitgezählt.

35:Declarations are not allowed here.

In C++ gelten Deklarationen zwar auch als Anweisungen, hier aber (z. B. direkt und ohne umschließende Blockklammern hinter `"if"` oder `"else"`) wird eine Anweisung, die keine Deklaration ist, erwartet.

36:Unions may not have members with Constructors or Destructors

Member von Unionen dürfen keine Konstruktoren oder Destruktoren besitzen (Tutorial 2.5.5 Ende).

37:A global anonymous union must be declared "static".

Deklariert man eine namenlose globale Union, also mehrere statische Objekte mit identischem Speicher, muß diese aus Gründen der Konsistenz und um den Linker nicht unnötig zu verwirren mit der Speicherklasse `"static"` geschehen. So ist nun mal die Regel, tut mir leid.

38:Functions can not be defined local to functions

Funktionsdefinitionen können in C (im Gegensatz z. B. zu Pascal) nicht ineinander verschachtelt werden. Vielleicht haben Sie bei der vorherigen Funktionsdefinition auch nur die schließende Klammer vergessen.

39:Function parameters differ from first declaration.

Da es in C kein Überladen von Funktionen gibt, kann der Compiler mit diesen widersprüchlichen Angaben über die Paraleterliste nicht leben. Im C++-Modus werden Sie dieser Fehlermeldung nicht begegnen.

40:";" expected.

Ein Semikolon `";"` fehlt - eine Fehlermeldung, die so ziemlich überall auftreten kann.

41:Incomplete type-specification.

Der Datentyp ist nicht hinreichend definiert, insbesondere ist z. B. seine Größe unbekannt. Wahrscheinlich versuchen Sie an dieser Stelle gerade, ein Objekt einer noch nicht definierten Klasse oder Struct zu definieren.

42:Vectors of references, functions or void are illegal.

Referenzen, Funktionen und `"void"` können nicht Elementtyp eines Vektors sein.

43:Vector has element type of unspecified size.

Der Datentyp der Vektorelemente (und damit auch der Vektortyp selbst) hat einen Speicherplatzbedarf, der dem Compiler momentan nicht bekannt ist. Möglicherweise habdelt es sich hier um eine bisher undefinierte Klasse.

44:Pointers/references to references are illegal.

Zeiger oder Referenzen auf Referenzen, also z. B. `"int &&"` oder `"int *&"`, sind nicht erlaubt.

45: Illegal referenced type.

Referenzen auf Funktionstypen und ähnlich abgefahrene Dinge gibt's nicht.

46: Function has illegal return type.

Der Ergebnistyp der Funktion ist ungeeignet, z. B. ein Vektor oder eine Funktion.

47: Declaration expected.

Hier erwartet der Compiler irgendeine Deklaration.

48: Struct/class "<xxx>" undefined.

Die angegebene Klasse ist zwar dem Namen nach bekannt (also deklariert), aber nicht definiert.

49: ")" expected.

An dieser Stelle würde der Compiler gern eine schließende Klammer sehen.

50: Identifier "<name>" redefined.

Der Bezeichner wurde im selben Gültigkeitsbereich schon einmal definiert, und zwar anders.

51: Definition has illegal Type.

Variablen und Member dürfen als Typ nicht `"void"`, eine undefinierte Klasse oder etwas ähnlich Sinnloses haben.

52: Syntax error in type description.

Ein Syntax-Fehler wurde in einem Konstrukt entdeckt, das der Compiler für eine Typbeschreibung hält.

53: Please switch to C-Mode for old style declarations.

Parameterlisten im alten "K&R"-Stil erlaubt MaxonC++ nur im ANSI C-Modus.

54: No actual "this" pointer.

Die Variable `"this"` gibt es hier nicht. Man kann sie nur in nicht-statischen Memberfunktionen benutzen.

55: Friend declaration: "<xxx>" unknown.

Wird in einer `"friend"`-Deklaration ein qualifizierter Bezeichner benutzt, z. B `"friend int C::f()"`, muß dieser Bezeichner (insbesondere diese Funktion) bereits deklariert sein, und zwar exakt typgleich.

56: This overloaded function has different access modes.

Zugriffs-Deklarationen sind nur für solche überladene Funktionsnamen erlaubt, bei denen sämtliche Funktionen gleichen Zugriffsmodus haben (Tutorial 3.2.2.3 Ende).

57: Member function re-declared.

Die Memberfunktion wurde nach Ende der Klassendefinition erstmals oder anders als bisher definiert.

58: Tried to change functions access.

Bei der Deklaration einer Member-Funktion wurden widersprüchliche Angaben über den Zugriffsmodus gemacht.

59: Return type differs form first declaration.

Bei einer vorhergehenden Funktionsdeklaration wurde ein anderer Ergebnistyp angegeben. Könnte auch sein, daß sie zuvor versehentlich ohne Prototyp aufgerufen wurde, der Compiler einen solchen mit Default-Ergebnistyp `"int"` eingeführt hat und nun ein anderer Typ deklariert wird.

60: Identifier expected.

Hier wird ein Bezeichner erwartet. Im Gegensatz zur wortgleichen Fehlermeldung Nummer 5 wird diese vom Parserteil des Compilers ausgeworfen.

62: Syntax error in enumeration.

Ein Syntx-Fehler in der Definition eines Aufzählungtyps.

63: Unions may not be derived.

Unionen dürfen keine Basisklassen haben. Könnte aber auch sein, daß der Doppelpunkt hinter dem Unions-Namen einfach ein Versehen ist.

64: ":" expected.

Das Symbol `":"` fehlt hinter Schlüsselworten wie `"private"` oder `"public"` auf.

65: No access to member "<name1>" of class "<name2>".

Auf den Member oder sonstigen Klasseninternen Bezeichner `<name1>` der Klasse `<name2>` darf hier nicht zugegriffen werden.

66: Can not define qualified identifier.

Nach der Definition einer Klasse wurde scheinbar versucht, einen bisher unbekannten Member bzw. eine Memberfunktion hinzuzufügen, was natürlich nicht statthaft ist. In der Regel haben wir es hier aber bloß mit einem Schreibfehler zu tun.

67: Data type undefined.

Die Größe dieses Datentyps ist bisher nicht bekannt, weshalb er hier nicht in dieser Weise benutzt werden kann.

68: Illegal use of storage class specifier.

Die angegebene Speicherklasse (z. B. `"auto"`) ist hier (z. B. in einer Klassen- bzw. Strukturdefinition oder auf Dateiebene) sinnfrei.

69: Class members can not get initialized.

Member von Klassen, auch nicht statische, dürfen innerhalb der Klassendefinition nicht initialisiert werden.

70: Syntax error in base-list.

In der Basisklassenliste einer Klasse wurde ein Fehler (das "syntaktisch" ist hier nicht so ernst zu nehmen, es kann auch ein semantischer sein) gefunden.

71: Multiple use of same base class.

Jede Klasse kann jede direkte Basisklasse nur einmal haben. Die bemäkelte Klasse ist in der Basisklassenliste aber schon einmal aufgetaucht.

72: Illegal use of "friend" specifier.

Hier (z. B. vor einer Variablen- oder Typdefinition) ist keine `"friend"`-Spezifizierung erlaubt.

73: Illegal use of Bitfields.

Mischungen von Bitfeldern und gewöhnlichen Membern sind in MaxonC++ nicht erlaubt (Tutorial 2.5.6).

74: Attempt to change access of member "<xxx>" of class "<yyy>".

Bei einer Zugriffs-Deklaration (Tutorial 3.2.2.3) wurde versucht, den Zugriffsmodus eines Members zu verändern.

75: Anonymous union: Illegal member "<xxx>".

Eine namenlose Union darf keine Member-Funktionen, statische Member und dergleichen haben, was auch irgendwie logisch ist (Tutorial 2.5.5 Ende).

76: Register objects must be local to a function.

Die Speicherklasse `"register"` ist nur innerhalb von Funktionen zulässig.

77: Syntax error in declaration.

Auf vielfältige Weise kann hier in dem, was der Compiler für eine Deklaration hält, ein falscher Fehler stecken.

78: Access to variable that was constructed under condition.

Tritt eine Variablendeklaration in einer Bedingung oder als Schleifenrumpf auf, ohne mit `{ ... }` geklammert zu sein, ist sie außerhalb des Schleifen- oder Abfragekonstrukts zwar prinzipiell sichtbar, darf aber nicht benutzt werden. Beispiel: Bei der Anweisung `"while (i) int j = i++;"` darf `"j"` später nicht mehr benutzt werden.

79: Syntax error in expression or declarator.

Wegen der konzeptionellen Ähnlichkeit von bestimmten Ausdrücken und Deklarationen, und weil in C++ manchmal erst in einer sehr späten Phase der Analyse zwischen Ausdrücken und Deklarationen unterschieden werden kann, wird deren Syntax teilweise schon geprüft, wenn der Compiler noch nicht weiß, was er vor sich hat. Immerhin hat er schon zu diesem Zeitpunkt feststellen können, daß es sich um einen syntaktischen Fehler handelt.

80:Constant integer expression expected.

Hier wird (z. B. als Elementanzahl eines Vektors) ein konstanter ganzzahliger Ausdruck erwartet.

81:"]" expected.

Die schließende Klammer "]" zu einer entsprechenden öffnenden Klammer fehlt - das könnte aber auch Folgefehler eines anderen Syntax-Errors sein.

82:Initialized object is not an aggregate.

Das Objekt, das hier mit einer Elementliste "{ ... }" initialisiert werden soll, ist kein Aggregat, also kein Vektor (Tutorial 2.1.2.6) und keine geeignete Struktur, die keine Basisklassen, Konstruktoren oder private Memberbesitzt (Tutorial 2.5.3).

83:"}" expected.

Die schließende Klammer "}" zu einer entsprechenden öffnenden Klammer fehlt - das könnte aber auch Folgefehler eines anderen Syntax-Errors sein.

84:Too many Vector elements given

Bei einer Listeninitialisierung wurden mehr Werte angegeben als der Vektor Elemente hat.

85:Constant String is too long

Ein Zeichenvektor wurde mit einem unzulässig langen String initialisiert. Beachten Sie hier auch den kleinen Unterschied zwischen C und C++ (Tutorial 2.1.2.5.2).

86:Initializer has illegal Type.

Der Initialisierungswert paßt vom Typ her nicht.

87:Too many struct/class members.

Bei einer Listeninitialisierung wurden mehr Werte angegeben als die Struktur Member hat.

88:Bitfield member has illegal size

Ein Member eines Bitfelds darf maximal 32 Bit breit sein, und natürlich muß die Bit-Breite auch eine echt positive Zahl sein.

89:Identifier "<xxx>" not defined.

Der angegebene Bezeichner ist bisher unbekannt. Es wird angenommen, daß es sich dabei um eine Variable handelt.

90:Member of Class "<xxx>" expected.

Hier, z. B. hinter "." oder "->", wird ein Member der genannten Klasse erwartet.

91:Access to ambiguous base class

Es wurde auf einen nicht-statischen Member oder eine nicht-statische Memberfunktion zugegriffen, die im Ableitungsbaum der vorliegenden Klasse mehrfach auftritt, wodurch dem Ausdruck kein eindeutiges Objekt zugeordnet werden kann.

92: Function call: can not use register A4 in small data model.

Das Register A4 wird bei Übersetzung im Small Data-Modell dringend anderweitig benötigt und kann deshalb nicht zur Parameterübergabe an Funktionen benutzt werden.

93: Subscribt: int expression in []'s expected.

Der Indexausdruck beim Vektorzugriff muß ganzzahlig sein.

94: Subscript: operand before []'s must be vector or pointer.

Vor dem mit "[ ]" geklammerten Index muß ein Zeiger- oder Vektorausdruck stehen.

96: Illegal type conversion

Diese Typumwandlung ist noch nicht einmal durch einen Cast möglich.

97: Operators "++" and "--" need a l-value.

Der Operand der unären Operatoren "++" und "--" muß ein L-Wert sein (Tutorial 1.4.3.2, 1.4.3.3).

98: Unary operator used on illegal type.

Ein unärer Operator wurde auf einen absolut unpassenden Ausdruck angewendet.

100: Address operator "&" needs a l-value.

Der unäre "&"-Operator kann ausschließlich auf L-Werte (also solche Ausdrücke, die z. B. auch links vom Zuweisungsoperator "=" stehen können) angewandt werden. Bitfeld-Member sind in diesem Zusammenhang keine L-Werte.

101: Function call: Expression left of () is not a function.

Der Compiler hält ein Klammerpaar "( )" (das nicht notwendig leer sein muß) für eine Argumentliste, den Ausdruck links davon aber nicht für eine Funktion.

103: Assignment: Type conflict.

Nur eine aufgeblähte Verrsion der Fehlermeldung #128 ("Type conflict"), die anzeigt, daß die nicht zueinander passenden Datentypen diesmal auf den beiden Seiten einer Wertzuweisung ("=") stehen.

104: Syntax error in Parameter list

In der Parameterlisten-Deklaration steckt ein syntaktischer Fehler.

105: Parameter has illegal Type.

Der Typ dieses Funktionsparameters ist unzulässig, z. B. könnte er bisher nicht hinreichend definiert sein.

106: Unknown function "<xxx>".

Zur dieser Funktion existiert kein Prototyp.

107: Default parameter has illegal type.

Der Typ des hier angegebenen Default-Arguments paßt nicht zum Parameter.

108: Double use of parameter register.

Das hier explizit angegebene Register ist in der vorliegenden Parameterliste schon anderweitig vergeben.

109: Redefinition of default argument for parameter "<xxx>".

Default-Argumente dürfen jeweils nur genau einmal spezifiziert werden (Tutorial 1.6.5.2).

110: Default Argument not at end.

Default-Argumente dürfen nur für die letzten Parameter einer Funktion definiert werden, d. h. es darf kein Parameter ohne Defaultargument mehr folgen (Tutorial 1.6.5.2).

111: Redefinition of default argument.

Default-Argumente dürfen jeweils nur genau einmal spezifiziert werden (Tutorial 1.6.5.2).

112: Argument missing.

Beim Funktionsaufruf wurde mindestens ein Argument zu wenig angegeben.

113: Ambiguous function call.

Beim vorliegenden Aufruf eines überladenen Funktionsnamens lassen die Argumente keinen eindeutigen Schluß auf die gewünschte Funktion zu (Tutorial 4.1.2).

114: Illegal argument.

Dieses Funktionsargument hat einen unpassenden Datentypen.

115: Function declaration: Illegal storage class.

Eine Funktion darf als Speicherklasse zwar `"extern"` oder `"static"`, nicht aber `"auto"`, `"register"` oder sonstwas Bescheuertes besitzen.

116: Illegal use of function specifier.

Die Funktions-Spezifizierung (`"inline"` oder `"virtual"`) ist hier nicht zulässig.

117: "inline" used after definition or use of function.

Die Funktion wird hier erstmals als "inline" deklariert, wurde zuvor aber schon definiert oder aufgerufen.

118: Function redefined.

Der Anweisungsteil dieser Funktion wurde bereits definiert, oder sie wurde schon einmal als Amiga-Library-Funktion deklariert.

119: "inline" used with storage class specifier.

Eine `"inline"`-Funktion darf weder "extern" noch "static" deklariert sein.

120: Inconsistent linkage.

Die Speicherklasse einer Funktion oder eines Objekts wurde zuvor bereits anders deklariert, und die vorliegende neue Deklaration widerspricht dem.

121: Local function definitions can only be "extern".

In C++ dürfen Funktiosdeklarationen, die lokal in Funktionen auftreten, aus Gründen der Eindeutigkeit ausschließlich als "extern" definiert werden.

122: Syntax error.

Der wohl vielseitigste Fehler überhaupt - dem Compiler geht hier ziemlich die Orientierung ab.

123: Unknown linkage.

Maxon C++ kennt die Link-Spezifikationen "C", "C++", "Asm", "Pascal" und "Amiga", aber das, was hier steht, nicht.

124: Linkage specification may only occur in file scope.

Link-Spezifikationen dürfen nicht lokal in Funktionen, sondern nur auf Dateiebene stehen (Tutorial 2.2.1.2.3).

125: Pointer to member: Class expression expected.

Der Typ des linken Operanden von `".*"` bzw. `"->*"` stimmt nicht: Es muß sich um eine Klasse bzw. einen Zeiger darauf handeln.

126: Illegal pointer-to-member expression.

Typfehler: mindestens ein Operand von `".*"` bzw. `"->*"` hat einen unpassenden Typ.

127: Assignment: Left value must be L-Value.

Bei einer Wertzuweisung mit `"="` muß der linke Wert ein L-Wert sein - was denn auch sonst, denn schließlich kommt Begriff "L-Wert" auch genau daher (Tutorial 1.4.3.2, 1.4.3.11 et al.).

128: Type conflict.

Diese Typkonvertierung ist zumindest implizit nicht möglich.

129: Can't convert "qualified type*" to plain "type*".

Für einen Datentyp `"T"` kann ein Zeiger auf `"const T"` oder `"volatile T"` nicht in einen gewöhnlichen Zeiger auf `"T"` ungewandelt werden, denn das würde durchs Hintertürchen diese beiden Typqualifizierungen ad absurdum führen (Tutorial 2.2.3.1).

130: Access to private base class.

Es wurde, insbesondere bei einer impliziten oder expliziten Typumwandlung, auf eine private Basisklasse zugegriffen, wobei die allgemeinen Zugriffs-Kontroll-Regeln verletzt wurden.

131: Member function call: class expression has illegal type.

Beim Aufruf einer Member-Funktion ist der Ausdruck links vom `"."` bzw. `"->"` vom falschen Typ, z. B. `"const"` oder `"volatile"`.

133: Conversion "int" -> "enum": allowed in C mode only.

In C++ ist es nicht möglich, einen ganzzahligen Wert ohne Cast in einen Aufzählingstyp umzuwandeln (Tutorial 2.4.3).

134: Operator or type description expected.

Hinter dem Schlüssenwort `"operator"` sollte entweder ein überladbarer Operator (dann ist es eine Operator-Funktion) oder eine Typbeschreibung stehen (dann ist es eine Konvertierungsfunktion).

135: Conversion functions must be member of a class.

Da Konvertierungsfunktionen ein Klassenobjekt in einen anderen Datentyp umwandeln, sonst aber keine Parameter besitzen, können Sie ausschließlich als nichtstatische Member einer Klasse deklariert werden.

136: Conversion functions may not be static.

Eine Konvertierungsfunktion ist zwar eine relativ gewöhnliche Memberfunktion, darf aber nicht als statisch deklariert werden. Kein Wunder, schließlich soll sie ja ein Klassenelement in einen anderen Typ konvertieren und hat keine expliziten Parameter, und deshalb steht sie ohne den impliziten "this"-Parameter ziemlich arm da.

137: Ambiguous conversion function.

Die hier explizit genannte Konvertierungsfunktion gibt es in mehreren Basisklassen, weshalb sie nicht eindeutig ist. Diese Fehlermeldung ist im Grunde genommen analog zu Error #176 (`"<id1>"` defined in classes `"<id2>"` and `"<id3>"`), nur daß wir es hier nicht mit einem wirklichen Bezeichner, sondern dem Pseudo-Namen einer Konvertierungsfunktion zu tun haben.

138: Conversion function not found.

Die genannte Konvertierungsfunktion wurde nie zuvor deklariert.

139: Type conversion by ambiguous user conversion functions

Für die gewünschte Typkonvertierung stehen mehrere Konvertierungsfunktionen zur Verfügung, ohne daß klar ist, welche hier zu benutzen ist.

140: Operator overload: at least one parameter must be "class".

Wenn Sie einen Operator überladen, muß wenigstens einer der Operanden von irgendeinem Klassentyp sein.

141: Operator overload: Illegal number of formal parameters.

Die hier deklarierte Operatorfunktion hat zu viele oder zu wenige Parameter.

142: Operators may not have default arguments or ellipsis.

Operator-Funktionen dürfen keine Default-Argumente und erst recht keine Ellipse `"..."` haben.

143: Second parameter must be "int".

Bei dieser Operatorfunktion (z. B. `"++"` oder `"--"`) muß der optionale zweite Operand den Typ "int" haben.

144: Conversion functions may not have an explicit return type.

Der Ergebnistyp einer Konverierungsfunktion ist durch ihren Namen (operator <typbeschreibung>) bereits implizit definiert. Es darf bei der Deklaration sonst kein Ergebnistyp angegeben werden.

145: Operators must not be static members.

Wenn eine Operatorfunktion schon als Member deklariert wird, sollte sie nicht statisch sein. Wahrscheinlich stammen Sie aus Benhausen.

146: Unary operator "<op>" used on illegal argument.

Der genannte unäre Operator wird mit einem unzulässigen Operanden benutzt.

147: Binary operator "<op>" used on illegal arguments

Ein binärer Operator wurde auf Operanden angewandt, die auch unter Berücksichtigung aller Überladungen einfach nicht passen wollen.

148: "." and "->" need a class expression as left operand

Der Ausdruck links des Operators `"."` oder `"->"` muß eine Klasse, Struktur, Union oder etwas, was sich mittels einer Konvertierungsfunktion darin verwandeln läßt, sein. Beachten Sie bitte auch die Möglichkeit, daß der Fehler in Mißachtung der Bindungsregeln bestehen und sich durch ein zusätzliches Klammerpaar beheben lassen könnte.

149: This operator must be a member function.

Bestimmte Operatoren, z. B. `"="`, können auschließlich als Memberfunktionen überladen werden.

150: Empty source!

Der Quelltext ist, jedenfalls nachdem er den Proprozessor passiert hat, völlig leer - formal zwar erlaubt, im Zweifelsfall aber wohl eher Folge eines Versehens. Zumindest können wir uns sicher darauf einigen, daß ein leeres Programm kein sinnvolles ist.

151: Operator "sizeof": Type has undefined size.

Die hier mit `"sizeof"` erfragte Größe des Datentyps ist auch dem Compiler selbst bisher noch nicht bekannt.

152: "sizeof" needs an expression or a type in "()"'s

Der Ausdruck bzw. die Typbeschreibung hinter dem `"sizeof"` ist nicht korrekt (Tutorial 1.4.3.3).

153: Destructor: class name expected after "~".

In dem, was der Compiler hier für einen Destruktor-Namen hält, muß hinter der Tilde `"~"` der aktuelle Klassenname stehen.

154: Declaration: can not use a "( )" initializer here.

Der Compiler hat eine Initialisierung in Klammern `"( )"` gefunden (oder glaubt es zumindest), welche hier aus irgendwelchen Gründen, z. B. in einem `"typedef"` oder einer Funktionsdeklaration, nicht erlaubt ist.

155: No matching constructor.

Es gibt keinen passenden Konstruktor, der die vorliegende Klasse mit dem angegebenen Ausdruck initialisiert.

156: No default constructor for class "<name>".

Die genannte Klasse hat keinen Default-Konstruktor, weshalb hier eine Initialisierung erforderlich ist. Der Hinweis kann auch am Ende der Definition einer Klasse stehen, wenn sie einen Member oder eine Basisklasse eines solchen Typs, aber keinen Konstruktor hat.

157: Ambiguous: use constructor or user conversion?

Der Compiler kann sich hier nicht recht entscheiden, ob er bei einer Typkonvertierung von einem Konstruktor oder einer Konvertierungsfunktion Gebrauch machen soll (Tutorial 3.5.2, vorletztes Beispiellisting)

158: Destructor declaration: Empty parameterlist "()" expected.

Ein Destruktor darf keine Parameter haben, die leere Parameterliste muß aber trotzdem angegeben werden.

159: Destructor name does not match class name.

Ein Destruktor muß so heißen wie die Klasse, für die er definiert wird.

160: Name of baseclass or nonstatic member expected.

Innerhalb einer Konstruktor-Initialisierungs-Liste wird hier der Name einer Basisklasse oder eines nicht-statischen Members erwartet (Tutorial 3.3.3).

161: Ctor initializer: "(" expected.

In der Initialisierungsliste eines Konstruktors wird hier ein rundes Klammerpaar mit dem Initialisierungswert darin erwartet.

162: Initializer for Member "<name>" expected.

Der Member `<name>` benötigt eine Initialisierung, weil er z. B. eine Referenz, ein konstantes Objekt oder ein Klassenobjekt ohne Defautkonstruktor ist.

163: Member or baseclass initialized twice.

In einer Konstruktor-Initialisierungs-Liste taucht der Member bzw. die Basisklasse doppelt auf (Tutorial 3.3.3).

164: Member "<name>" not initialized.

Der Member `<name>` benötigt eine Initialisierung, weil er z. B. eine Referenz, ein konstantes Objekt oder ein Klassenobjekt ohne Defautkonstruktor ist.

165: Operator "new" must return a pointer.

Der Ergebnistyp eines überladenen `"new"`-Operators muß ein Zeigertyp sein - strenggenommen sogar ein Zeiger auf `"void"`, aber da ist MaxonC++ etwas toleranter als nötig (Tutorial 4.2.2.2.2).

166: Operator "delete" must return "void".

Überlädt man den Operator `"delete"`, muß er einen Ergebnistyp haben, und zwar `"void"` und sonst nichts (Tutorial 4.2.2.2.3).

167: Operator "new": type expected.

Hinter `"new"` wird ein Datentyp erwartet (Tutorial 2.6.2).

168: This type does not represent an object.

Mit `"new"` dürfen nur Objekte erzeugt werden - und z. B. Referenzen, Funktionen oder "void" sind in diesem Sinne keine.

169: Vector element type has undefined size.

Der Datentyp der Vektorelemente (und damit auch der Vektortyp selbst) hat einen Speicherplatzbedarf, der dem Compiler momentan nicht bekannt ist.

170: Virtual functions differ in return type.

Es wurde versucht, mit `"new"` ein Objekt eines Typs bisher unbekannter Größe zu erzeugen.

171: Allocation of vectors: no initializer allowed.

Wird mit `"new"` ein Vektor erzeugt, kann dieser nicht explizit initialisiert werden. Auch bei Vektoren von Klassen findet nur eine implizite Initilisierung durch den Default-Konstruktor statt.

172: Operator "delete": 1st parameter must have type "void *".

Der erste Parameter eines überladenen Operators `"delete"` kann ausschließlich vom Typ `"void*"` sein (Tutorial 4.2.2.2.3).

173: Operator "delete": 2nd parameter must have type "size_t".

Wenn ein `"delete"`-Operator zwei Parameter hat, muß der zweite den Typ `"size_t"`, und das ist in Maxon C++ "unsigned int", haben (Tutorial 4.2.2.2.3).

174: Operator "delete": too many parameters

Mehr als zwei Parameter darf der `"delete"`-Operator nicht haben (Tutorial 4.2.2.2.3).

175: Can not overload operator "delete".

In jeder Klasse kann nur ein `"delete"`-Operator definiert sein (Tutorial 4.2.2.2.3).

176: "<id1>" defined in classes "<id2>" and "<id3>".

Der Member `<id1>` tritt in den sich gegenseitig nicht überdeckenden Basisklassen `<id2>` und `<id3>` der vorliegenden Klasse auf und kann deshalb nicht eindeutig zugeordnet werden.

177: Virtual functions must be members of a class.

Natürlich können virtuelle Funktionen nur Memberfunktionen einer Klasse sein.

178: Virtual functions can not be static.

Eine virtuelle Memberfunktion ist naturgemäß an ein ganz bestimmtes Objekt gebunden und kann deshalb niemals als "static" deklariert werden.

179: Class "<id1>" is abstract (pure virtual function "<id2>" in baseclass "<id3>").

`<id1>` ist eine abstrakte Klasse und kann als solche hier nicht benutzt werden (insbesondere dürfen auf keine Weise Objekte dieser Klasse definiert werden). Als besonderer Service wird darauf hingewiesen, daß `<id2>` eine der rein virtuellen Funktionen der ist und ursprünglich in der Basisklasse `<id3>` definiert wurde.

180: Virtual functions differ in return type.

Die Funktion überdeckt eine gleichnamige virtuelle Funktion in einer Basisklasse, hat aber einen anderen Ergebnistyp.

181: Class "<id1>" is abstract (pure virtual function "<id2>").

`<id1>` ist eine abstrakte Klasse und kann als solche hier nicht benutzt werden (insbesondere dürfen auf keine Weise Objekte dieser Klasse definiert werden). Als besonderer Service wird darauf hingewiesen, daß `<id2>` eine der rein virtuellen Funktionen der Klasse ist.

182: Constructor initializer: "{" expected.

Am Ende der Initialisierungsliste eines Konstruktors wird der mit `"{"` beginnende eigentliche Funktionsrumpf erwartet (Tutorial 3.3.3).

183: Calling pure virtual function.

Es ist schlechterdings unmöglich (und führt zu einem Linker-Fehler), eine als rein virtuell definierte Memberfunktion aufzurufen (Tutorial 3.2.3.3). Rein formal kann man es aber trotzdem versuchen, und zwar über einen qualifizierten Bezeichner oder innerhalb eines Konstruktors. Mit dieser Fehlermeldung haut der Compiler einem dann auf die Pfoten.

184: Pointer expression expected.

Hinter `"delete"` muß als Operand natürlich ein Zeigerausdruck stehen.

185: Can´t convert virtual base class to derived class.

Hat man einen Zeiger auf eine virtuelle Basisklasse, kann man ihn auch durch einen Cast nicht in einen Zeiger auf eine abgeleitete Klasse umwandeln, denn in diese Richtung gibt es keinen Zeiger (Tutorial 3.2.5, Beispiel mit `"VirtBasis"` und `"VirtAbgeleitet"`).

186: Reference conversion: L-value expected.

Die Referenz muß hier mit einem geeigneten L-Wert initialisiert werden. Der Compiler kann hier also kein temporäres Objekt einführen.

187: Can not subtract pointers of different types.

Die mit "-" berechnete Differenz zweier Zeigerausdrücke ist nicht etwa die Differenz der Speicheradressen, sondern die Anzahl der dazwischen liegenden Vektorelemente. Deshalb können ausschließlich Zeiger auf den selben Typ voneinander subtrahiert werden.

188: Referenced type has unknown size.

Bei Additionen und Subtraktionen auf Zeigern (auch mit "++", "--", "+=", "-=") muß die Größe des vom Zeiger referierten Datentyps bekannt sein. Bei Zeigern auf Klassen und Strukturen muß diese z. B. definiert worden sein.

189: Division by zero.

In einem konstanten Ausdruck, der zur Compilezeit ausgewertet wurde, trat eine Division durch Null auf.

190: Operator requires integral operands.

Bei "<<", ">", "%" und anderen Operatoren sind keine Fließkomma-, sondern nur ganzzahlige Operanden erlaubt.

191: Attempt to assign to const object.

Sinn und Zweck eines als "const" deklarierten Objekts ist, daß ihm eben nach der Initailisierung kein neuer Wert mehr zugewiesen werden darf - und das wurde auf die eine oder andere Weise versucht (Tutorial 2.2.3.1).

192: No assignment to vectors, functions, and void expressions.

Vektoren, Funktionen und "void"-Ausdrücke können grundsätzlich nicht auf der linken Seite einer Wertzuweisung stehen.

193: Expressions left and right of ":" need common type.

Bei einem bedingten Ausdruck (Tutorial 1.4.3.10) müssen der "if"- und der "else"-Teilausdruck (also die Ausdrücke beiderseits des ":") einen gemeinsamen Datentyp haben oder sich zumindest ineinander umwandeln lassen können.

194: Conditional expression: ":" expected.

Ein bedingter Ausdruck erfordert hinter dem "?" stets gleichermaßen einen "if"- und einen "else"-Teil, die durch ein ":" zu trennen sind.

195: Condition expected.

Hier wurde eine logische Bedingung erwartet. Dabei kann es sich um einen beliebigen skalaren Ausdruck handeln.

196: Illegal parameter for "operator = ()".

Einem Klassenobjekt, auf dem der Operator "=" überladen ist, wird hier ein unpassender Ausdruck zugewiesen.

197: += / -= used on pointer: integral expr. as 2nd operand required.

Bei den Operatoren `"+="` und `"-="` mit einem Zeigerausdruck als linkem Operanden muß der rechte Operand (also das, was addiert werden soll) natürlich ganzzahlig sein.

198: Attempt to assign to undefined class.

Eine Struktur oder Klasse wurde zwar deklariert, z. B. `"class Hallo;"`, aber ihr Inhalt (=ihre Member) noch nicht definiert. Deshalb ist noch keine Wertzuweisung zwischen Objekten dieser Klasse möglich.

199: Label "<name>" redefined.

Ein Sprunglabel namens `<name>` gibt es in dieser Funktion schon.

200: Label expected.

Hinter `"goto"` wird ein Label erwartet, und das ist in C grundsätzlich ein Bezeichner.

201: Label "<name>" undefined.

Die Sprungmarke `<name>` wurde benutzt, aber nirgendwo gesetzt.

202: Jump past initialization of object "<name>".

Eine Sprunganweisung, insbesondere bei `"goto"` oder `"switch"`, aber unter bestimmten Umständen auch in Form einer Schleifenstruktur oder Fallunterscheidung, überspringt die Initialisierung der Variablen `<name>`.

203: "(" expected.

Hinter `"for"`, `"switch"` oder einem ähnlichen Schlüsselwort wird ein Klammernpaar erwartet.

204: "else" without "if".

Ein `"else"` ohne passendes `"if"` wurde gelesen.

205: "continue" used outside iteration statement.

Die Anweisung `"continue"` kann nur innerhalb einer geeigneten Schleifenanweisung gesetzt werden.

206: "break" without matching iteration or switch statement.

Die `"break"`-Anweisung darf nur innerhalb von geeigneten Schleifen oder `"switch"`-Statements auftreten (Tutorial 1.5.4.2).

207: "while" expected.

Am Ende der `"do"`-Schleife wird hier das `"while"` schmerzlich vermißt.

208: Function needs return value.

Da die Funktion einen Ergebnistyp ungleich `"void"` hat, wird hinter dem `"return"` ein passender Rückgabeausdruck erwartet.

209: Void functions may not return data.

Die Funktion hat den Ergebnistyp `"void"`, weshalb hinter `"return"` keinerlei Ergebnistyp (auch kein `"void"`-Ausdruck) stehen darf.

210: Expression does not match function return type.

Der Ausdruck hinter dem `"return"` paßt typmäßig nicht zum Ergebnistyp der Funktion und kann auch nicht dorthin umgewandelt werden.

211: Integral or enumerated expression expected.

Ein Ausdruck eines ganzzahligen oder Aufzählingstyps wird erwartet, insbesondere als Argument von `"switch"`.

212: "case" without "switch"

Ein `"case"`-Label ohne passendes `"switch"`-Statement wurde gefunden.

213: ":" expected.

Hinter einer `"case"`-Sprungmarke fehlt hier der Doppelpunkt.

214: Expression does not match type of "switch" expression.

Der Ausdruck hinter dem `"case"` läßt sich (jedenfalls implizit) nicht in den (ganzzahligen oder Aufzählungs-) Typ des Ausdrucks der `"switch"`-Anweisung umwandeln.

215: "case" requires constant expression.

Naturgemäß darf hinter einem `"case"`-Label ausschließlich ein konstanter skalarer Ausdruck stehen.

216: Constant appears twice in same "switch" statement.

Jeder konstante `"case"`-Wert darf innerhalb eines `"switch"` nur genau einmal auftreten.

217: "default" without "switch"

Genau wie `"case"` kann `"default"` nur innerhalb einer `"switch"`-Anweisung auftreten.

218: Each "switch" statement may have only one "default" label.

In jeder `"switch"`-Anweisung darf das Label `"default:"` höchstens einmal auftreten.

219: Old-style parameter declaration expected.

Innerhalb einer Parameterlistendefinition im alten Kernighan-Ritchie-Stil liegt hier ein (meist offensichtlicher) syntaktischer oder semantischer Fehler vor.

220: Can not overload old-style functions.

MaxonC++ weigert sich, bei Funktionen mit Parameterlisten im alten K&R-Stil das C++-Feature Überladung zuzulassen. Das ist ganz einfach zu blöd.

221: Can not take address of register parameter.

Wurde ein Parameter in MaxonC++ als `"register"` deklariert, darf seine Adresse weder direkt (mit dem unären `"&"`-Operator) noch indirekt (durch Setzen einer Referenz auf die Variable) genommen werden. Das ist auch einsichtig, denn es wurde zuvor der gesamten Umwelt groß und breit kundgetan, daß der Parameter beim Funktionsaufruf in ein Register geladen werden soll, und deshalb kann der Compiler die Variable schlecht noch nachträglich auf den Stack raufschieben.

222: Inconsistent use of register parameters.

Zwei Prototypen für ein und dieselbe Funktion unterscheiden sich hinsichtlich der Definition der Registerparameter.

223: Sorry, can't convert virtual base class to derived class.

Ein Zeiger auf eine virtuelle Basisklasse kann nicht wieder in einen Zeiger auf die abgeleitete Klasse umgewandelt werden (Tutorial 3.2.5).

224: Function "<xxx>" has internal linkage and is undefined.

Die Funktion `<ident>` wurde als `"static"` deklariert, aber nirgendwo im Programm definiert. Da sie offenbar nicht über den Linker eingebunden werden kann, ist dies ein Fehler.

225: Name of pointer/int variable expected.

Innerhalb einer Amiga-Linkspezifikation wird hier der Name der Basisvariablen, die (sofern überhaupt schon deklariert) bitteschön `"int"` oder von einem Zeigertyp sein sollte (Tutorial 2.2.1.2.4).

226: Library vector Offset ("= <short constant>") expected.

Bei der Linkspezifikation `"Amiga"` wird hinter jeder Funktionsdeklaration ein `"="` und der ganzzahlige Offset erwartet (Tutorial 2.2.1.2.4).

227: Implicit conversion "void*" -> "TYPE*": allowed in C Mode only.

Nur in ANSI C kann ein Zeiger auf `"void"` implizit (d. h. ohne Cast und dergleichen) in einen anderen Zeigertypen umgewandelt werden. C++ erwartet hier eine explizite Typkonvertierung (Tutorial 2.1.1.4).

228: Can not mix declarations and other statements in C mode.

In ANSI C müssen innerhalb eines Blocks sämtliche Variablendeklarationen vor der ersten Anweisung stehen (Tutorial 1.5.2). Das "other" steht hier deshalb, weil eine Deklaration in C++ ebenfalls als Anweisung gilt.

229: Sorry, but ANSI C does not know member functions...

In ANSI C gibt es definitiv keine Member-Funktionen.

230: Syntax error in #pragma

In einem MaxonC++ bekannten `#pragma` wurde ein Fehler gefunden.

231: "amicall": non-overloaded function expected.

Die `"#pragma amicall"`-Deklaration darf sich nur auf Funktionen beziehen, für die bereits ein Prototyp gelesen wurde, und ist für überladene Funktionen auch nicht erlaubt (Tutorial 5.5.3).

232: "amicall": int/pointer variable expected.

Innerhalb eines `"amicall"`-Pragmas wird hier die Basisvariable, nämlich ein Bezeichner für eine globale `"int"`- oder Pointervariable, erwartet (Tutorial 5.5.3).

233: Function already used.

Für eine Funktion, die zuvor bereits ein "gewöhnlicher" Aufruf stattgefunden hat, wird hier nachträglich ein `"#pragma amicall"` angegeben, was nicht erlaubt ist.

234: Every reference needs an initializer.

Referenz-Objekte müssen grundsätzlich initialisiert werden.

235: Can not take address of a "register" object.

Es ist in C nicht erlaubt, die Adresse eines als `"register"` deklarierten Objekts zu nehmen, nicht einmal dann, wenn das Objekt sowieso nicht in ein Register geladen wird. C++ ist da toleranter: wenn die Adresse der Variablen irgendwo auftaucht, wird die Variable eben nicht in einem Register abgelegt. Mit "Adresse nehmen" ist insbesondere die Anwendung des unären Operators `"&"` gemeint, aber auch implizite Adressermittling, z. B. wenn eine Referenz mit dem Objekt initialisiert wird (Tutorial 2.2.1.3).

236: Syntax error inside inlined member function "<xxx>".

Inline-Memberfunktionen, also solche kurzen Funktionen, die direkt in der Klassendefinition selbst definiert werden, werden zunächst nur sehr grob (durch Zählen der Klammern) analysiert, um erst nach Ende der Klassendefinition endgültig geprüft zu werden. Bei dieser ersten groben Analyse wurde ein offensichtlicher syntaktischer Fehler gefunden.

237: Unexpected end of source.

Der Quelltext endet vorzeitig, z. B. mitten in einer Funktions- oder Klassendefinition.

238: Automatic variable used outside of its function.

In C++ ist es tatsächlich möglich, Funktionen ineinander zu verschachteln, indem man nämlich lokale Klassen mit Memberfunktionen deklariert. In diesen Memberfunktionen sind auch die Bezeichner der umschließenden Fnktion sichtbar - aber die Variablen dürfen trotzdem nicht angerührt werden, weil dies enorme implementatorische Schwierigkeiten verursachen würde.

239: Can't reference "const TYPE" with plain "TYPE &".

Diese Fehlermeldung ist im Grunde genommen das Gegenstück zu Error #129 (Can't convert "qualified type*" to plain "type*"), nur mit Referenzen statt Zeigern.

240: A template declaration may only appear as a global declaration.

Templates sind generell global. Sie dürfen nur auf Dateiebene, also keinesfalls in einer Klasse oder Funktion, deklariert und definiert werden.

241: Operator "new": first operand must be of type "size_t".

Der erste Parameter einer `"new"`-Operatorfunktion muß vom Typ `"size_t"`, also `"unsigned int"` in der vorliegenden Implementierung, sein (Tutorial 4.2.2.2.2).

242: Template argument list: "<" expected.

Die Argumentliste eines Templates beginnt stets mit der öffnenden spitzen Klammer `"<"`, und genau dies wird hier vermißt.

243: Template argument list: ">" expected.

Das Ende einer Template-Argumentliste bildet die schließende Klammer. Der Compiler ist der Überzeugung, daß die Argumentliste jetzt zu Ende ist, und erwartet deshalb das `">"`.

244: Template argument expected.

In der Template-Deklaration wird hier ein Typ- oder Wertparameter erwartet.

245: Template argument list differs from previous declaration.

Templates sind nicht (wie Funktionsnamen) überladbar. Folglich muß die Parameterliste eines Templates bei jeder Deklaration oder Definition praktisch identisch sein. Allenfalls die Namen der Argumente dürfen differieren.

246: This template has already been defined.

Es wird soeben versucht, ein bereits prima definiertes Template noch einmal zu definieren.

247: Syntax error inside class template "<xxx>".

Klassentemplates werden zunächst nur sehr grob auf syntaktische Korrektheit geprüft (genaugenommen werden hauptsächlich die Klammern gezählt), und die eigentliche syntaktische und semantische Überprüfung erfolgt erst, wenn das Template benutzt wird. Hier wurde aber schon bei der ersten Prüfung ein Fehler entdeckt.

248: Type argument expected.

Als Argument (bei einer Templatebenutzung) wird an dieser Stelle eine Typbeschreibung erwartet.

249: Template argument list: "," expected.

Die Argumentliste des Templates ist noch keineswegs zuende. Deshalb werden ein Komma und noch mindestens ein weiteres Argument erwartet.

250: Constant expression as template argument expected.

Wenn ein Template ein Wert-Argument erwartet, muß dieses auch konstant sein. Pointer auf statische Objekte gehen notfalls aber auch durch.

251: Template: class or function declaration expected.

Templates gibt es in C++ für Klassen und Funktionen und sonst für nix.

252: Syntax error inside template function "<name>".

Funktionstemplates werden zunächst nur sehr grob auf syntaktische Korrektheit geprüft (genaugenommen werden hauptsächlich die Klammern gezählt), und die eigentliche syntaktische und semantische Überprüfung erfolgt erst, wenn aus dem Template eine Funktion erzeugt wird. Hier wurde aber schon bei der ersten Prüfung ein Fehler entdeckt.

253: Function templates: Argument "<name>" never used.

Alle Parameter (und das sind stets Typparameter) eines Funktionstemplates müssen aus Gründen der Eindeutigkeit in der Parameterliste der Funktion auftauchen. Ein Auftreten im Ergebnistyp oder Anweisungsteil der Funktion zählt in diesem Zusammenhang nicht.

254: No such member function in class "<name>".

Die hier deklarierte Member-Funktion tritt in der genannten Template-Klasse nicht auf. Wahrscheinlich gibt es sie sogar in überhaupt keiner aus diesem Template generierten Klasse.

255: Template function redefined.

Das Funktionstemplate war zuletzt mit anderen Parametern oder einem anderen Ergebnistyp deklariert worden.

256: Arguments do not match function template.

Bei der Benutzung eines Templates passen die hinter dem Templatenamen in spitzen Klammern `"<...>"` angegebenen Argumente nicht.

257: Class name collides with template name.

Auf Dateiebene oder an sonstwie unpassender Stelle trat eine Typangabe wie `"class XXX"` auf, wobei `"XXX"` Name eines Klassentemplates ist, aber keine Template-Argumentliste folgt. Dies ist nicht zulässig.

258: Inconsistent exception specification

Wenn ein Prototyp einer Funktion mehrfach aufgeführt wird, darf die Ausnahme-Deklaration jedesmal wiederholt werden - man kann es aber auch lassen. Wenn man sich die Mühe macht, müssen die Deklarationen aber natürlich übereinstimmen.

259: "{" expected.

Try-Blöcke und Catch-Handler sind stets Verbundanweisungen und müssen deshalb in geschweifte Klammern eingeschlossen werden.

260: "catch" expected.

Hinter einem Try-Block muß mindestens eine Catch-Answeidsung folgen. Sonst hätte das "try" ja auch ziemlich wenig Sinn.

261: "throw": Expression expected.

Hinter `"throw"` kann entweder garnichts oder ein Ausdruck stehen. Eine Typangabe z. B. reicht nicht.

262: Illegal jump into try block.

Mit einem `"goto"`-Sprung wurde in einen Try-Block hineingesprungen, was der Compiler nicht erlaubt.

263: No "#pragma header" found.

Wenn Sie mit der Option `"-h"` bzw. der entsprechenden Einstellung in einem Dialogfenster eine vorcompilierte Header-Datei benutzen wollen, muß Ihr Quelltext eine Zeile `"#pragma header"` enthalten. Diese muß in der Haupt-Quelldatei und nicht etwa in einer Includedatei stehen.

264: Can not open header file "<name>".

Die vorcompilierte Header-Datei kann nicht geöffnet werden. Diese Meldung kann beim Lesen oder Schreiben der Datei auftreten.

265: Header file is invalid!

Das, was mit der Option `"-h"` bzw. der entsprechenden Einstellung in einem Dialogfenster als Header-Datei geladen werde sollte, ist gar keine.

266: Header file has wrong version number - please recompile it!

Die vorcompilierte Headerdatei konnte zwar geladen werden, aber sie wurde mit einer anderen Compilerversion erzeugt und paßt deshalb nicht zum momentan benutzten Compiler.

267: "#pragma header" may appear on file scope only.

Die Zeile `"#pragma header"` darf nur zwischen den Deklarationen auf Datei-Ebene auftreten, also insbesondere nicht innerhalb von Funktions- oder Klassendefinitionen und auch nicht mitten in einer globalen Deklaration.

B-3. Sonstige Fehlermeldungen

Nachdem der Compiler den Analyseteil abgeschlossen und `"No errors"` gemeldet hat, ist aber noch nicht alles gelaufen, was potentiell schiefgehen kann. Da gibt es zum Beispiel theoretisch die folgende Meldung:

"Internal error in Code generation"

Ich hoffe sehr, daß dieser Fehler bei Ihnen niemals auftritt. Da MaxonC++ paranoid programmiert wurde, prüft es an gegebener Stelle öfters, ob das, was es da macht, überhaupt so sein kann. Kommt es dabei zu dem Schluß, daß intern irgendetwas höllisch schiefgelaufen sein muß,

macht es sich benerkbar - im Analyseteil mit der Fehlermeldung `"Compiler is in Panic!"`, während der Codeerzeugung mit "Internal Error...".

Was tun, wenn so ein Fehler trotzdem auftritt? Melden Sie sich umgehend bei MAXON, am besten mit einer möglichst detaillierten Fehlerbeschreibung, denn es handelt sich hier um einen Bug des Compilers und liegt nicht unbedingt an Ihrem Programm, auch wenn die Chancen gut stehen, daß ein solcher Fehler durch ein Übermaß an Fehlermeldungen provoziert wurde.

Als "erste Hilfe" können Sie versuchen, herauszufinden, welche Stelle Ihres Programms die Ursache war: Lassen Sie sich mit der Option `"-as"` bzw. dem ***"Source in Asm"***-Gadget ein kommentiertes Assemblerlisting erzeugen, und sehen Sie darin nach, an welcher Quelltextposition die Codererzeugung abgebrochen wird (da endet das Assembler-Listing). Vielleicht fällt Ihnen ja etwas ein, wie Sie die betreffende Stelle unverfänglicher programmieren können.

Wie gesagt, ich hoffe, daß der oben genannte Fehler bei Ihnen wirklich niemals unverschuldet auftritt. Eine zweite Fehlermeldung, die ebenfalls während der Codegenerierung auftreten kann, ist weit weniger problematisch:

"Workspace overflow."

Diese Worte wollen Ihnen ganz einfach sagen, daß der Compiler-Arbeitsspeicher zu knapp bemessen ist. Erhöhen Sie diese Einstellung einfach mit der Option `"-w"` bzw. dem entsprechenden Gadget im Compiler-Requester nach Gutdünken und starten Sie den Compiler erneut.

AMIGA

MaxonC++

Tutorial

MAXON
computer

Einführung in C++

Ein paar kurze Bemerkungen, bevor es los geht

Dieses Buch soll eine Einführung in das Programmieren mit MaxonC++ darstellen. Es wäre sehr nützlich, wenn Sie bereits eine Programmiersprache beherrschen, und wenn es bloß AmigaBASIC ist. Sonst werde ich nichts als bekannt voraussetzen. Wenn Sie sogar schon C kennen, werden Sie den Anfang dieses Handbuchs wahrscheinlich nur überfliegen, denn C++ basiert auf C, und wir werden uns zunächst mit ziemlich elementaren Dingen beschäftigen, bei denen es kaum Unterschiede zwischen C und C++ gibt.

Damit wären wir auch schon bei einem kleinen Problem: Eigentlich ist dies eine Einführung in gleich drei Programmiersprachen: C, C++ und MaxonC++. Der Grund dafür liegt in dem Chaos, das es bei den diversen Sprachstandards gibt: Da wäre zum einen der ANSI-C-Standard, den MaxonC++ vollständig enthält. C++ ist von ANSI C abgeleitet (genaugenommen ist es noch etwas komplizierter: C++ entstand aus der „alten" C-Sprachdefinition und ANSI C enthält außer jenem Standard einige Features aus C++, was aber nichts an der Tatsache ändert, daß ANSI C im wesentlichen eine Untermenge von C++ ist). Einen verbindlichen ANSI-Standard für C++ gibt es derzeit noch nicht, nur den de-fakto-Standard, der vom C++-Erfinder Bjarne Stroustrup festgelegt wurde und auch „AT&T"-Standard genannt wird. Davon gibt es aber verschiedene Versionen, von denen eine als „AT&T 2.0" bzw. „2.1" (das ist im Prinzip dasselbe) bekannt ist. Darauf basiert MaxonC++.

Jeder Sprachstandard enthält Dinge, die implementationsabhängig sind, d. h. der Compilerbauer hat gewisse Freiheiten. Dies ist meist durch die verschiedenen Rechnerarchitekturen erforderlich. Deshalb muß dieses Handbuch auch Auskunft darüber geben, wie die jeweiligen Details in MaxonC++ implementiert sind. Außerdem hat MaxonC++, wie wohl jeder Compiler, kleinere Features, die über die Sprachdefinition hinausgehen. Auch diese müssen im Handbuch natürlich dokumentiert werden.

Wenn ich also im folgenden von „C" oder „ANSI C" spreche, so gilt das gesagte jeweils auch für C++ und somit für MaxonC++, sofern ich mich nicht ausdrücklich auf die Unterschiede zwischen diesen drei Programmiersprachen beziehe.

1. Der Einstieg

1.1 Ein erstes Beispiel

An dieser Stelle könnte ich viele Worte über das objektorientierte Programmieren und andere wichtige und nützliche Konzepte von C++ und die Vorzüge und die Leistungsfähigkeit dieser Programmiersprache im allgemeinen und von MaxonC++ im besonderen verlieren. Entweder wissen Sie schon, was „objektorientiert" heißt, und dann würden Sie diesen Abschnitt sowieso überlesen, oder Sie kennen es noch nicht, und dann würde ein allgemeiner Exkurs zu diesem Thema bei Ihnen erfahrungsgemäß doch eher Ratlosigkeit und Verwirrung als Klarheit und Erkenntnis schaffen.

Und so ziehe ich es vor, ohne weitere Vorrede gleich zur Einführung in die Programmiersprache C++ überzugehen, und verspreche Ihnen, Sie im folgenden mit Satzungetümen wie den obigen zu verschonen. Also gleich das erste Beispiel:

```
// Das unvermeidliche Hallo-Programm
#include <stream.h>

void main()
{ cout « "Hallo!";}
```

Am besten tippen Sie das Programm in Editor ab, starten den Compiler und lassen das Programm dann laufen, so wie es im ersten Teil dieses Handbuchs erklärt wurde. Dann sollte das Programm den Text **„Hallo!"** ausgeben.

Falls das Compilieren oder Linken nicht geklappt haben sollte, kann das unterschiedliche Ursachen haben. Zunächst könnte natürlich der Compiler einen Fehler melden, falls Sie das Programm nicht richtig abgetippt haben. Auch kann es sein, daß der Compiler die Includedatei *„stream.h"* nicht findet, weil Sie das C++-System nicht korrekt installiert haben, oder daß der Linker aus demselben Grund eine Library nicht finden kann. Dann sollten Sie sich den entsprechenden Abschnitt dieses Handbuchs noch einmal ansehen.

Nehmen wir also an, daß Sie das Prograämmchen zum Laufen gebracht haben. Was bedeutet das Ganze nun?

Die erste Zeile ist ein Kommentar. Der Compiler erkennt das an dem doppelten Slash `"//"` am Zeilenanfang. Ganz allgemein können Sie auch an jede Zeile eines C++-Programms auf diese Weise einen Kommentar anhängen. Der Compiler ignoriert diese Kommentare vollständig, sie dienen nur dazu, das Programm dem menschlichen Leser - also z. B. Ihnen - zu erläutern.

Es gibt noch eine andere Schreibweise für Kommentare, die eine Art Erbschaft von der Programmiersprache C ist:

```
/* Auch dies ist ein Kommentar. Er beginnt mit einem "/" und einem "*"
und kann sich dann über beliebig viele Zeilen  erstrecken. Beendet
wird er dann mit diesen beiden Zeichen: */
```

Diese Kommentare enden also nicht automatisch am Zeilenende, sondern werden mit der Zeichenfolge `"*/"` abgeschlossen. Sie sind eigentlich nur dann praktisch, wenn man einen längeren Text als Kommentar in den Programmtext setzen will. In der Praxis hat man es aber meist mit kürzeren Kommentaren zu tun, und da ist die `"//"`-Schreibweise einfach bequemer. Außerdem kann man daran auf den ersten Blick erkennen, daß man ein C++- und nicht etwa ein C-Programm vor sich hat. Selbst hartnäckige Gegner der objektorientierten Programmierung können nicht leugnen, daß diese Kommentare eine enorm sinnvolle Neuerung von C++ sind.

Die zweite Zeile beginnt mit einem `"#"`. Solche Zeilen werden genaugenommen nicht vom Compiler selbst, sondern vom sog. Preprozessor bearbeitet. Dieser Preprozessor ist ein Programm, durch das der Quelltext noch vor dem eigentlichen Compilieren läuft. Neben vielen anderen nützlichen Funktionen, auf die ich in Kapitel 5 eingehen werde, entfernt er Kommentare und führt Zeilen aus, die mit eben jenem `"#"` beginnen. Das Kommando `"include"` weist ihn an, an Stelle dieser Zeile den Inhalt einer Datei in den Quelltext einzufügen - hier ist es eine Datei namens *„stream.h"*.

Natürlich löscht der Preprozessor die Kommentare nicht wirklich aus ihrem Quelltext, und er fügt Include-Dateien auch nicht tatsächlich in den Sourcecode ein. Damit ist gemeint, daß er eine entsprechend modifizierte Kopie des Quelltexts erzeugt, die dann vom Compiler übersetzt wird. In Wirklichkeit wird in MaxonC++ gar keine echte Kopie angelegt (das würde nur Speicher verschwenden), aber lassen wir das und tun einfach so als ob.

Die Datei *„stream.h"* enthält einige Definitionen, die für die Ein- und Ausgabe nahezu unverzichtbar sind. Deshalb werden Sie diese Include-Datei wohl in den meisten Programmen verwenden.

Nach dem Kommentar für den Leser und der Preprozessor-Anweisung beginnt nun unser eigentliches Programm.

Die Zeile

```
void main()
```

bedeutet folgendes:

- Es wird eine Funktion definiert (gekennzeichnet durch das leere Klammerpaar `"()"`).
- Der Name dieser Funktion ist `"main"`. Das ist eine ganz besondere Funktion, die in keinem Programm fehlen darf: Sie stellt das Hauptprogramm dar.
- Die Funktion hat keinen Rückgabewert, bzw. einen Wert des leeren Typs `"void"`. Was das bedeutet, werden Sie bei passender Gelegenheit erfahren.

Als nächstes folgt der Inhalt oder „Rumpf" dieser Funktion. Er ist mit geschweiften Klammern `"{"` und `"}"` einzuschließen und enthält hier nur eine einzige Anweisung:

```
cout « "Hallo!";
```

Das Wort `"cout"` bezeichnet ein Datenobjekt, das in der Includedatei `"stream.h"` deklariert wird - deshalb brauchten wir also die `"#include"`-Zeile. Was ein Datenobjekt im allgemeinen ist

und wie das Objekt `"cout"` genau aussieht und ob ein Objekt wirklich irgendwie „aussieht" oder ob das nur so eine Redensart ist und viele Fragen mehr, möchte ich an dieser Stelle erst einmal zurückstellen. Sie sollten sich zunächst einmal damit begnügen, daß man in C++ mit einer Anweisung der Art

```
cout « Ausdruck;
```

einen Ausdruck auf den Bildschirm ausgibt, in unserem Fall war der Ausdruck der Text **"Hallo!"**. Jede Anweisung endet in C++ mit einem Semikolon.

1.2 Lexikalisches

An dieser Stelle sind noch einige ganz allgemeine Anmerkungen über die Gestalt von Programmen in C++ nötig:

Zunächst ist zu erwähnen, daß in C++ die Groß-/Kleinschreibung von Namen generell signifikant ist. Mit der Zeile

```
Void MAIN()
```

hätte der Compiler also nichts anfangen können. Hier unterscheidet sich C++ von Programmiersprachen wie Pascal oder dem allseits beliebten AmigaBasic.

Als Ausgleich für diese Pingeligkeit ist der Compiler aber in bezug auf die Textformatierung absolut tolerant. Es ist ihm ziemlich egal, wie Sie Ihr Programm in Zeilen aufteilen, wie weit Sie Zeilen einrücken, wo Sie Leerzeichen oder Kommentare einfügen usw. Unsere Funktion `"main"` hätten Sie also auch so schreiben können:

```
void main (
  ){/* Dies ist ein Kommentar */cout
 « /* */ "Hallo!" // Noch ein Kommentar
 ; } // Hier ist das Programm zuende. Grüße an Oma!
```

...oder so:

```
void main(){cout«"Hallo!";}
```

Für Menschen wie Sie und mich ist das natürlich nicht so besonders leserlich, aber der Compiler betrachtet ein Programm ausschließlich als eine Folge von Eingabesymbolen, deren textliche Anordnung keine Rolle spielt. Wichtig ist nur, daß er die Symbole voneinander trennen kann, z. B. ist deshalb eine Lücke zwischen `"void"` und `"main"` unentbehrlich. Im Gegensatz dazu achtet der Preprozessor durchaus auf die richtige Zeilenaufteilung: Eine Preprozessor-Anweisung beginnt stets mit einem `"#"`, das in der ersten Spalte einer Zeile stehen muß, und ist immer genau eine Zeile lang.

Da wir gerade beim Thema sind, können wir auch gleich die unterschiedlichen Symbole betrachten, aus denen sich ein Programm zusammensetzt.

Zunächst wären da die Wortsymbole. In unserem kleinen Beispielprogramm benutzen wir nur das Wortsymbol (oder Schlüsselwort) `"void"`. Eine vollständige Übersicht über alle Schlüsselwörter von C++ finden Sie im Anhang dieses Handbuchs.

Etwas scheinbar Ähnliches, aber in Wahrheit ganz anderes sind die Bezeichner (Identifier). Das sind vom Benutzer definiertbare Namen für Objekte, Funktionen, Datentypen und einiges mehr. Im *„Hallo"*- Programm benutzen wir den Bezeichner `"cout"`, der in `"stream.h"` deklariert wurde, und definieren selbst den Bezeichner `"main"` als Namen einer Funktion.

Ein Bezeichner ist eine beliebig lange Folge von Buchstaben, Ziffern und dem Unterstrich `"_"`. Es darf aber keine Ziffer am Anfang stehen. Gültige Bezeichner wären also z. B.

```
main
TeSt
Eumel26731
Dies_ist_ein_ziemlich_langer_Name
_1_
x
```

Außerdem gibt es diverse Literale, das sind Konstanten mit einem bestimmten Wert. Wir haben bisher nur eine Zeichenketten-Konstante benutzt, nämlich `"Hallo!"`. Eine solche Zeichenkette (String) wird mit doppelten Anführungszeichen eingeschlossen und kann beliebig viele Zeichen enthalten. Sie darf aber ein Zeilenende nicht überschreiten. Wenn man mehrere Zeichenketten hintereinander schreibt, hängt der Preprozessor sie aneinander, so daß die Anweisung

```
cout « "Dies" "sind" "mehrere Zeichenketten"".";
```

identisch mit

```
cout « "Diessindmehrere Zeichenketten.";
```

ist.

Man kann mit dem Backslash `"\"` besondere Zeichen in einen solchen String einschließen.

Hier sollen nur zwei erwähnt werden: Die Folge `"\n"` wird in einem String durch das Linefeed-Zeichen ersetzt, bewirkt also bei der Ausgabe einen Zeilenvorschub.

Ein Beispiel:

```
cout « "Life! Don't talk\nto me about life!";
```

gibt

Life! Don't talk
to me about life!

aus. Sie können doppelte Anführungszeichen in einen String einschließen, indem Sie einen Backslash voranstellen:

```
cout « "Written by Jens \"Himpelsoft\" Gelhar 1994.";
```

schreibt

Written by Jens "Himpelsoft" Gelhar 1994.

Außerdem gibt es noch etliche Literal-Schreibweisen für Zahlen, die wir im nächsten Kapitel kennenlernen werden.

Die vierte Kategorie von Eingabesymbolen sind die Sonderzeichen: Diverse eckige, runde und geschweifte Klammern, Operatoren wie z. B. `"«"`, Trenner wird das Semikolon `";"` oder das Komma `","` und einige andere, die wir noch kennenlernen werden. Auch dazu gibt es eine vollständige Übersicht im Anhang.

1.3 Zahlen, Variablen und Berechnungen

Nun können Sie also schon per Programm Texte ausgeben. Bekanntlich kann ein Computer aber wesentlich besser mit Zahlen umgehen, und genau darum soll es in diesem Kapitel gehen.

Außerdem benötigt man zum Programmieren auf jeden Fall Variablen. Eine Variable ist in C, genau wie in allen anderen real existierenden Programmiersprachen auch, ein Datenobjekt eines festgelegten Typs, das unterschiedliche Werte enthalten kann. Man definiert einfache Variablen in C nach folgendem Schema:

```
Typname Variablenname, Variablenname, ..., Variablenname;
```

Am besten betrachten wir gleich ein Beispielprogramm, im dem ein paar Variablen und andere neue Sprachelemente vorkommen:

```
#include <stream.h>

void main()
{  // Variablen muß man definieren, bevor man sie benutzt:
   int Eingabe1, Eingabe2;
   int Summe, Produkt, Mittelwert;

   // Jetzt kommen die Anweisungen unseres Programms:
   cout « "Bitte geben Sie zwei Zahlen ein: ";

   // Die folgende Anweisung erwartet zwei Eingaben des Benutzers:
   cin » Eingabe1 » Eingabe2;

   // Nun haben diese beiden Variablen Werte, und wir können
   // einige Berechnungen durchführen:
   Summe = Eingabe1+Eingabe2;
   Produkt = Eingabe1*Eingabe2;
   Mittelwert = Summe/2;

   // Wir geben die Ergebnisse aus:
   cout « "\n"          // Zeilenvorschub, sieht einfach besser aus
   « "Die Summe von " « Eingabe1 « " und " « Eingabe2
```

```
  « " ist " « Summe « ".\n";
  cout « "Das Produkt dieser Zahlen ist " « Produkt « ".\n";
  cout « "Ihr arithmetisches Mittel ist " « Mittelwert
  « " (abgerundet).\n";
}
```

Das Programm fängt ganz gewohnt an: Wir benutzen wieder die Definitionen aus `<stream.h>` und deklarieren die Hauptfunktion `"main"`. Der Anweisungsteil beginnt dann auch sofort mit zwei Zeilen mit Variablendefinitionen. Wie Sie aus dem zuvor gesagten messerscharf schließen können, ist hier `"int"` jeweils der Name des Datentyps.

Eine Variable des Typs `"int"` kann eine ganze Zahl enthalten. Diese kann positiv oder negativ sein, darf aber nicht beliebig groß oder klein werden, nämlich so ungefähr im Bereich von `-2000000000` bis `+2000000000` (in Worten: plus/minus zwei Milliarden). Es gibt in C++ noch weitere Zahl-Datentypen, die z. B. noch größere oder auch gebrochene oder nur positive Zahlen umfassen. Ich möchte diese aber lieber in einem späteren Kapitel im Überblick einführen, denn für unsere ersten Programmierversuche reicht der Datentyp `"int"` vollkommen aus.

Die erste `"int"`-Zeile definiert also zwei Variablen eben diesen Typs. Ihre Namen - hier `"Eingabe1"` und `"Eingabe2"` - sind Bezeichner, die wir frei wählen können, einmal abgesehen davon, daß wir natürlich die im vorherigen Abschnitt genannten Regeln für Bezeichner beachten müssen und daß wir jeden Namen nur einmal verwenden können. Wir wählen hier die Namen so, daß man daran sofort erkennen kann, was die Variablen zu bedeuten haben, daß sie nämlich für Eingaben des Benutzers gedacht sind. Wenn das aus Längengründen nicht möglich ist - und das ist bei komplexeren Programmen der Normalfall - sollten Sie hinter die Definition einen Kommentar setzen, der die Bedeutung der Variablen erläutert.

Wichtig

In dem Moment, wo eine Variable definiert wird, hat sie noch keinen (bzw. einen zufälligen, undefinierten) Wert!

Die zweite `"int"`-Zeile definiert drei weitere Variablen, ebenfalls mit aussagekräftigen Namen. Es hat keinen besonderen Grund, daß wir die Definition hier auf zwei Zeilen aufgeteilt haben, genauso gut hätten wir auch alle fünf Variablen in einer Zeile definieren können. Auch müssen die Variablendefinitionen in C++ nicht immer am Anfang einer Funktion stehen: es reicht, wenn sie vor ihrer ersten Benutzung definiert werden.

Übrigens: `"int"` ist kein Bezeichner, sondern ein Wortsymbol wie `"void"`. Und auch `"void"` bezeichnet einen Datentypen - allerdings den leeren Typen, der keinerlei Werte umfaßt und auch gar keine irgendwie besonderen Eigenschaften hat. Deshalb ist es auch nicht erlaubt, eine Variable des Typs `"void"` zu definieren. Der Typ `"void"` kann nur zu ganz bestimmten Zwecken eingesetzt werden, z. B. als Pseudo-Rückgabetyp einer Funktion, die gar kein Ergebnis liefert.

Nun folgen also die Anweisungen unseres Programms. Deren erste verstehen Sie ja bereits: Der Benutzer wird aufgefordert, zwei Zahlen einzutippen. Neu ist hingegen `"cin"`. Das ist, genau wie

`"cout"`, ein Datenobjekt, nur steht es für die Standardeingabe statt -ausgabe. Der Operator `"»"`, angewandt auf das Objekt `"cin"` und eine Variable, liest einen Wert des passenden Typs - hier also Int-Zahlen - vom Standard-Eingabemedium und weist ihn der Variablen zu. Also wartet das Programm hier, bis der Benutzer zwei Zahlen eingegeben hat, und weist diese dann den beiden Variablen zu.

Natürlich kann nicht nur der Anwender (durch eine Texteingabe) einer Variablen einen Wert zuweisen - das kann das Programm auch selbst. Dazu dient in C++ der Operator `"="`.

Wie Sie sehen, steht linkes vom `"="` immer eine Variable und rechts ein Ausdruck, der der Variablen zugewiesen werden soll. Ein Ausdruck besteht in C++ im wesentlichen aus Variablen, Konstanten und Operatoren. Dabei funktioniert alles, wie Sie es aus der Mathematik gewohnt sind: `"+"` addiert, `"*"` multipliziert, `"/"` dividiert, es gilt die Regel „Punktrechnung vor Strichrechnung", und Sie können bei Bedarf auch Klammern setzen, und zwar runde: `"()"`. Zu erwähnen ist noch, daß der Divisionsoperator, wenn er auf ganze Zahlen angewendet wird, auch ein ganzzahliges Ergebnis liefert, indem er bei der Division auftretende Nachkommastellen einfach abschneidet. Außer den in unserem Programm benutzten Operatoren gibt es noch das `"-"` für die Subtraktion, das auch als Vorzeichen verwendet werden kann, und den Modulo-Operator `"%"`, der den bei einer Division auftretenden Rest liefert, und noch jede Menge andere.

Nach den drei Zuweisungsoperationen haben also unsere drei Variablen geeignete Werte, die dann in drei `"cout"`-Zeilen ausgegeben werden. Hier haben wir mehrfach den Zeilenvorschub `"\n"` benutzt, um zu erreichen, daß jeder Wert in eine eigene Zeile ausgegeben wird. Wir hätten natürlich auch alle drei `"cout"`'s zu einem einzigen Ausdruck zusammenfassen können, aber so bleibt es wenigstens noch halbwegs übersichtlich.

1.4 Numerische Datentypen und Operatoren

1.4.1 Ganzzahlige Datentypen

Den wohl wichtigsten und meistgebrauchten numerischen Typen kennen Sie ja schon, nämlich `"int"`. Dieser Datentyp ist, wie man so schön sagt, 32 Bit „breit", wobei ein Bit für das Vorzeichen gebraucht wird. Der Wertebereich geht damit von `-2147483648` bis `+2147483647` (daß der Bereich im negativen Bereich um 1 größer ist, kommt indirekt daher, daß es nicht sinnvoll ist, zwischen `+0` und `-0` zu unterscheiden. Die redundante `-0` wird gewissermaßen zur Codierung einer zusätzlichen negativen Zahl benutzt).

Oft hat man es in Programmen aber auch mit Zahlen zu tun, die gar nicht negativ werden können, und dann wäre es doch Verschwendung, ein Bit für das Vorzeichen zu benutzen. Außerdem ist es doch auch viel schöner, wenn man den Umstand, daß man es mit positiven Zahlen zu tun hat, im Programmtext ausdrücken kann. Deshalb gibt es auch eine vorzeichenlose Variante von `"int"`, naheliegenderweise `"unsigned int"` genannt.

Einmal davon abgesehen, daß der Typname hier aus zwei Schlüsselwörtern besteht, deklariert man eine vorzeichenlose Variable genauso wie ein gewöhnliches `"int"`, z. B. so:

```
unsigned int IrgendNeVariable;
```

- wobei man auch das `"int"` weglassen kann:

```
unsigned NochNeVariable;
```

Der Wertebereich solcher Variablen geht dann von `0` bis `4294967295`.

Nicht immer - genaugenommen sogar recht selten - hat man es mit großen Zahlen zu tun. Oft kann man Speicherplatz sparen, wenn man schmalere Datentypen benutzt. Unter bestimmten Umständen spart das sogar Rechenzeit, da der normale 68000-Prozessor ja eigentlich nur ein 16-Bit-Prozessor ist. Also gibt es die Datentypen `"short int"` sowie `"unsigned short int"`, die nur 16 Bit breit sind und einen entsprechend geringeren Wertebereich haben, den Sie der Tabelle auf der übernächsten Seite entnehmen können. Auch hier ist es Ihnen wie auch bei den folgenden Datentypen freigestellt, das `"int"` einfach wegzulassen.

Die genauen Datenbreiten sind leider in keinem Standard definiert, es gibt nur Mindestwerte. Deshalb gibt es auch C-Compiler, die allerdings langsam aussterben, mit nur 16 Bit breiten `"int"`-Typen. Die wirklich großen Datentypen heißen dann `"long int"` bzw. `"unsigned long int"`. MaxonC^{++} kennt diese Datentypen natürlich auch, aber sie sind eben nur genauso lang wie die entsprechenden `"int"`-Typen. `"int"` und `"long int"` sind aber trotzdem formal zwei verschiedene Datentypen und nicht nur zwei verschiedene Schreibweisen für denselben Typen!

Dafür hat MaxonC^{++} eine kleine Besonderheit: 64 Bit breite Zahlen! Ursprünglich war nämlich vorgesehen, die `"long"`-Typen 64 Bit breit zu wählen, aber aus Kompatiblitätsgründen wurde dann davon abgesehen. Statt dessen gibt es jetzt die Datentypen `"long long int"` und `"unsigned long long int"`, die ca. 19-stellige Zahlen aufnehmen können - ein nettes Feature, das man wohl nur selten brauchen wird, aber was man hat, das hat man. Es sei hier aber vor gedankenloser Benutzung dieser Datentypen gewarnt: Der Prozessor kann intern nur mit 32-Bit-Werten rechnen (und extern, wie oben angedeutet, sogar nur 16 Bit lesen und schreiben). Deshalb muß ein Programm für so ziemlich jede 64-Bit-Operation eine Bibliotheksfunktion aufrufen, entsprechend langsam geht das dann.

Übrigens: Man kann nicht nur das `"int"` am Ende optional weglassen. Es ist auch erlaubt, bei den vorzeichenbehafteten Datentypen `"signed"` davorzusetzen, etwa `"signed int"` (was nichts anderes als `"int"` ist) oder `"signed long long"`.

Für Programmierer, die von Pascal oder ähnlichen Sprachen auf C umsteigen, ist es oft befremdlich, daß ein Zeichen in C nichts anderes als eine Zahl ist. Das ist im Prinzip natürlich vollkommen korrekt, denn der Computer kann nun einmal ausschließlich mit Zahlen arbeiten, so daß es intern keinen Unterschied zwischen einer 8 Bit breiten Zahl und einem Zeichen gibt. Deshalb kann man in C^{++} (Ästheten würden hier „leider" hinzufügen) den Datentypen `"char"` sowohl für Zahlen als auch für Zeichen verwenden - kommt ganz darauf an, wie man es betrachtet.

Folgerichtig gibt es auch Zeichen (oder sollte man besser „Ein-Byte-Zahlen" sagen?) mit und ohne Vorzeichen, also die Datentypen `"signed char"` und `"unsigned char"`. Die Wertebereiche können Sie wieder der Tabelle am Ende dieses Abschnitts entnehmen.

Aber ist `"char"` nun identisch mit `"signed char"` oder `"unsigned char"`? Der ANSI-C-Standard läßt diese Frage offen, und damit hier gar keine Verwirrung entsteht, ist festgelegt, daß die drei Datentypen alle verschieden sind. In der Praxis ist `"char"` in MaxonC++ aber identisch mit `"signed char"`.

So, und nun kommt endlich die versprochend Tabelle der Datentypen von MaxonC++:

Typ	von	bis
`signed char`	`-128`	`127`
`char`	`-128`	`127`
`unsigned char`	`0`	`255`
`short`	`-32768`	`32767`
`unsigned short`	`0`	`65535`
`int`	`-2147483648`	`2147483647`
`unsigned int`	`0`	`4294967295`
`long`	`-2147483648`	`2147483647`
`unsigned long`	`0`	`4294967295`
`long long`	`-9223372036854775808`	`9223372036854775807`
`unsigned long long`	`0`	`18446744073709551615`

1.4.2 Literale für ganze Zahlen und Zeichenketten

Ein „Literal" ist ganz allgemein eine vom Compiler akzeptierte Schreibweise für eine Konstante. In dem kleinen Programm

```
void main()
{ int i;
  i = 42;
}
```

ist `"42"` eine Schreibweise für die bekannte und beliebte Zahl 42. Das klingt natürlich noch nicht sehr aufregend, aber es gibt in C noch viele andere mögliche Schreibweisen für ganze Zahlen.

Zunächst kann man im Literal angeben, welchen Typ die Zahl haben soll. Ein angehängtes `"u"` oder `"U"`, auch „Suffix" genannt, macht aus der Zahl einen vorzeichenlosen Wert, während das Suffix `"l"` oder `"L"` den Datentypen `"long int"` erzwingt.

Beispiele:

```
42 ist int
42U  ist unsigned
42L  ist long
42UL ist unsigned long
```

MaxonC++ kennt außerdem noch die Suffixe "LL" und "ULL" für die ganz breiten Datentypen:

```
42LL ist long long
42ULL  ist unsigned long long
```

Außerdem gibt es in C noch verschiedene Zahlenbasen, nämlich Hexadezimal und Oktal. Hexadezimale Zahlen beginnen mit "0x" oder "0X" und erlaubte Ziffern sind '0' bis '9', 'a' bis 'f' bzw. 'A' bis 'F'. Wie Sie sehen, ist bei Zahlendarstellungen die Groß-/Kleinschreibung generell nicht signifikant. Zum Beispiel hat "0xABCD" den Wert 43981 und "0X2A" ist 42.

Weniger gebräuchlich sind inzwischen die oktalen Zahlen, also die mit der Basis 8. Deshalb ist ihre Schreibweise wohl eher ein Relikt aus der Frühzeit von C: oktal sind nämlich alle Zahlen, die mit einer Null beginnen. "052" ist also dezimal 42 und nicht etwa 52!

Immerhin ist man inzwischen so weit (seit ANSI-C), daß '8' und '9' keine oktalen Ziffern sind. In alten Zeiten war "08" eine gültige Schreibweise für 8 - das gibt es jetzt nicht mehr. Jetzt gehen die Ziffern von '0' bis '7', wie sich das gehört.

Bekanntlich gibt es in C keinen wirklichen Unterschied zwischen Zahlen und Zeichen, weshalb in diesem Abschnitt auch char-Literale behandelt werden sollen. Für Zeichen verwendet man einfache Anführungszeichen, z. B.

```
char c, d;
c = 'X';
d = c+1;
```

Als Ergebnis erhält "d" hier übrigens das Zeichen, dessen Code um 1 größer als der von 'X' ist, nämlich 'Y'.

Genau wie bei konstanten Zeichenketten kann man auch hier die nicht-darstellbaren Steuerzeichen benutzen:

```
char c;
c = '\n';
```

weist "c" das Zeilentrennerzeichen zu, das übrigens auch LF (Linefeed) genannt wird und den Code 10 hat. Es werden sowohl bei Zeichen- als auch bei Zeichenkettenkonstanten folgende Steuer- und sonstige Zeichen unterstützt:

```
\n Zeilentrenner (LF)
\t Tabulator
\b Backspace (ein Zeichen zurückgehen)
\r Wagenrücklauf (Cursor an Zeilenanfang setzen)
\f Seitenvorschub (Bildschirm löschen)
\a Klingelzeichen
\\ das Gegenschrägstrich-Zeichen selbst
\? Fragezeichen
\' einfaches Anfügrungszeichen (z. B. als Charkonstante '\'')
\" doppelte Anführungszeichen (z. B. in Stringkonstanten)
```

Das Fragezeichen kann man natürlich auch ohne vorgestellten Gegenschrägstrich in Strings verwenden. Es gibt in ANSI C allerdings die sog. Trigraph-Sequenzen, mit denen man den C-Standard auch auf Rechnern mit einem etwas seltsamen Zeichensatz mehr oder weniger benutzbar machen will. Dabei kann man jeweils zwei Fragezeichen, gefolgt von einem dritten Zeichen, als Ersatz für einige ASCII-Zeichen verwenden. Z. B. ist `"??="` eine Ersatzschreibweise für `"#"`, und `"??)"` kann man überall als Ersatz für die eckige Klammer `"]"` benutzen. Das braucht Sie auf Ihrem Amiga eigentlich nicht zu stören, denn Sie haben ja alle notwendigen Zeichen auf Ihrer Tastatur (auch wenn Sie evtl. manchmal etwas suchen müssen). Probleme gibt es höchstens dann, wenn Sie in einer Zeichenkette zwei Fragezeichen hintereinander verwenden und zufällig das falsche Zeichen dahinter setzen:

```
cout « "(Bahnhof??)";
```

gibt nicht etwa `"(Bahnhof??)"`, sondern `"(Bahnhof]"` aus. Wenn Sie aber vor das zweite Fragezeichen einen Gegenschrägstrich setzen, können Sie diese Ersetzung unterdrücken, denn der Compiler ersetzt erst Trigraph-Sequenzen durch ihre Entsprechungen und erkennt dann erst Sonderzeichenfolgen in Strings.

Wenn Sie ein ganz besonderes Zeichen wollen, können Sie auch den entsprechenden ASCII-Code in Strings einsetzen. Sie haben die Wahl zwischen oktaler und hexadezimaler Angabe des Zeichencodes: Hinter den Backslach setzen Sie entweder bis zu drei oktale Ziffern oder ein `"x"`, gefolgt von beliebig vielen Hexadezimal-Ziffern. Das Linefeed-Zeichen `"\n"` kann man also auch durch `"\12"` oder `"\x0a"` ersetzen, was natürlich die Portabilität des Programms unnötig einschränken würde. Allerdings ist es üblich (und auch legal), das Nullzeichen, das in C Strings abschließt, `'\0'` zu schreiben. Man könnte hier natürlich auch die Zahlenkonstante 0 verwenden, denn es gibt ja keinen Unterschied zwischen Zeichen und Zahlen, aber die erste Schreibweise ist anschaulicher, da klargemacht wird, daß hier wirklich das Zeichen 0 gemeint ist.

1.4.3 Ganzzahlige Operatoren und Berechnungen

1.4.3.1 Allgemeines

C und C++ bieten dem Programmierer eine reiche Auswahl an Operatoren. Die einfachsten arithmetischen Operatoren `"+"`, `"-"`, `"*"` und `"/"` sowie den Zuweisungsoperator `"="` haben Sie ja bereits kennengelernt. Hier gilt die aus der Mathematik bekennte Regel „Punktrechnung vor Strichrechnung", d. h. `1+2*3` entspricht `"1+(2*3)"`. Wenn Sie hier die Auswertungsreihenfolge ändern wollen, können Sie entsprechend (runde) Klammern setzen. Außerdem ist zu erwähnen, daß die Bindung von links nach rechts erfolgt: `"26731-47-11-42"` entspricht erwartungsgemäß `"((26731-47)-11)-42"` und nicht etwa `"26731-(47-(11-42))"`. Das ist insofern erwähnenswert, als daß es in C++ durchaus auch Operatoren gibt, bei denen die Bindung von rechts erfolgt.

Damit wir nicht völlig die Übersicht verlieren, möchte ich die Operatoren in der Reihenfolge ihrer Bindung vorstellen, d. h. zuerst die, die am stärksten binden.

1.4.3.2 Postfix-Operatoren

Vorerst interessieren uns hier nur die nachgestellten Inkrement- und Dekrement-Operatoren `"++"` und `"--"`. Sie bieten eine einfache und übersichtliche Möglichkeit, den Wert einer Variablen um 1 zu erhöhen bzw. zu erniedrigen.

Beispiel:

```
int i;
i = 42;
i++;
```

Anschließend hat `"i"` den Wert 43; bei `"--"` wäre dementsprechend 1 subtrahiert worden. Als Operanden benötigen `"++"` und `"--"` einen sogenannten L-Wert. Bisher sind Variablen die einzigen L-Werte, die wir bisher kennengelernt haben. Die Bezeichnung kommt übrigens daher, daß ein L-Wert insbesondere auf der linken Seite des Zuweisungsoperators `"="` stehen darf, so wie in `"i = 42"`. Der Ausdruck `"41++"` wäre also nicht erlaubt.

Jeder Operator hat in C++ ein Ergebnis. `"++"` und `"--"` liefern als Ergebnis den Wert, den der L-Wert vorher (!) gehabt hatte. Also ist folgende Codesequenz nicht unbedingt schön, aber möglich:

```
int i, j;
i = 42;
j = i--;
```

Anschließend ist `"i"` wieder 41, während `"j"` den alten Wert von `"i"`, also 42, erhält.

Das Ergebnis ist allerdings kein L-Wert. Folgende naheliegende Anweisung zum Addieren von 2 ist also nicht erlaubt:

```
i++ ++;
```

Da diese beiden Postfixoperatoren also nur höchstens einmal angewendet werden können, ist es eigentlich müßig, ob die Bindung nun von links oder von rechts erfolgt. Wir werden später aber noch andere Postfixoperatoren kennenlernen, und deshalb sei jetzt schon der Vollständigkeit halber erwähnt, das die Auswertung von links erfolgt - was auch einsichtig ist, denn wenn da obiges Beispiel erlaubt wäre, würde es natürlich als `"(i++)++"` ausgewertet, während `"i(++ ++)"` überhaupt keinen Sinn ergäbe.

Am Rande sei noch bemerkt, daß der Ursprung der Bezeichnung „C++" im Dunkeln liegt. Die wahrscheinlichste Deutung ist, daß hier der Inkrementoperator auf die Programmiersprache „C" angewandt wurde, wobei allerdings der Operator ein wenig unglücklich gewählt wurde, denn wie Sie jetzt wissen, ist das Ergebnis von „C++" ja der alte Wert von „C". Wesentlich aussagekräftiger wäre deshalb der Name „++C", womit wir auch schon bei der nächsten Gruppe von Operatoren sind:

1.4.3.3 Unäre Operatoren

Die Operatoren `"++"` und `"--"` können einem L-Wert auch vorangestellt werden. Dann wird zum Operanden ebenfalls 1 addiert bzw. subtrahiert, aber hier wird als Ergebnis der neue Wert des Operanden geliefert.

Beispiel:

```
int i, j;
i = 42;
j = ++i;
```

Nun sind sowohl "i" als auch "j" 43. Erfahrungsgemäß ist es für C-Anfänger gar nicht so einfach, diese beiden Anwendungsarten derselben Operatoren auseinander zu halten. Am besten prägen Sie sich ein, daß bei "++var" zuerst addiert und dann der Variablenwert gelesen wird, während es bei "var++" entsprechend umgekehrt ist (für "--" analog).

Vier weitere unäre Operatoren können ebenfalls auf ganzen Zahlen angewendet werden:

+ Das Vorzeichen "+" macht nichts mit dem dahinterstehenden Ausdruck und ist somit ziemlich überflüssig.

- Das Minus kann ebenfalls als Vorzeichen benutzt werden und liefert den negierten Wert des Operanden. Bei vorzeichenlosen Datentypen ist das Ergebnis theoretisch undefiniert. MaxonC++ bildet hier das sog. 2-Komplement der vorzeichenlosen Zahl, aber darauf kann man sich nicht bei allen Compilern verlassen.

! Dieser Operator wird allgemein „nicht" (bzw. "not") genannt und bildet die logische Negation des Operanden: !x ist 1, wenn x = 0 gilt, und sonst 1. Die Operation ist nicht umkehrbar, denn !!x ist nicht identisch mit x.

~ Die Tilde steht in C für bitweise Negation: Jedes einzelne Bit wird umgedreht, d. h. aus 1 wird 0 und umgekehrt. Bei vorzeichenbehafteten Werten gilt ~0 = -1, ~42 = -43. Der Operator ist übrigens seine eigene Umkehrung: ~~x ergibt in jedem Fall wieder x.

Da diese Operatoren (im Gegensatz zu "++" und "--") den Wert des Operanden nicht verändern, muß dieser kein L-Wert sein. Die Bindung erfolgt natürlich von rechts:

"--!++x" ist "-(~(!(++x)))". Man beachte auch, daß "-1" in C kein Literal, sondern ein Ausdruck mit dem Operator "-" und dem Literal "1" ist.

Unäre Operatoren binden schwächer als Postfix-Operatoren. Also ist der Ausdruck

```
-i--
```

erlaubt und entspricht

```
-(i--) ,
```

was übrigens ein Beispiel für die berüchtigte Lesbarkeit von C ist.

Ein weiterer unärer Operator ist "sizeof". Er wertet den Operanden nicht aus, sondern gibt nur an, wieviel Speicher er belegt. Das Ergebnis hat laut ANSI den Datentypen "size_t", der in der Includedatei <stdlib.h> definiert ist. Unter MaxonC++ ist "size_t" identisch mit "unsigned int".

Es gibt zwei verscheidene Anwendungsarten von `"sizeof"`:

- Der Operand ist kein Ausdruck, sondern der Name eines Datentyps. Dann ist er in runde Klammern zu setzen, etwa so:

```
sizeof(long int)
```

Dieser Ausdruck ergibt in MaxonC++ übrigens `4`, oder genaugenommen `4u`, denn das Ergebnis ist ja `"unsigned int"`. In C++ ist garantiert, daß `"sizeof(char)" 1` liefert.

- Als Operand wird ein Ausdruck angegeben, der in Klammern stehen darf, aber nicht muß. Dann wird als Ergebnis der Speicherplatzbedarf des Typs des Operanden geliefert. Da der Typ eines Ausdrucks in C++ schon beim Compilieren festgestellt wird, wird der Ausdruck zur Laufzeit des Programms gar nicht ausgewertet.

Beispiel:

```
int i, j;
i = 42, j = sizeof i++;
```

`"j"` erhält als Wert den Speicherplatzbedarf des Typs von `i++`, der identisch mit dem von `"i"` ist, nämlich `"int"`. In MaxonC++ ist `"sizeof(int)" 4`. Da der Operand `"i++"` aber nicht ausgewertet wird, wird der Wert von `"i"` auch nicht verändert - es gilt weiter `"i=42"`.

Der `"sizeof"`-Operator wird vor allem für maschinennahe „low level"- Operationen benötigt. In C ist er außerdem wichtig, damit man bei der Funktion `"malloc"` weiß, wieviel Speicher angefordert werden soll, aber daß geht in C++ viel schöner und eleganter. Dazu später mehr.

1.4.3.4 Multiplikative Operatoren

So heißen die drei Operatoren, die gleichen Vorrang wie `"*"` haben, nämlich `"/"`, `"%"` und natürlich `"*"` selbst.

Die Multiplikation `"*"` kennen Sie bereits. Zur Division `"/"` ist nur zu sagen, daß das Ergebnis bei ganzzahligen Operanden ebenfalls ganzzahlig ist und stets abgerundet wird. Ist einer der Operanden negativ, so werden die Beträge dividiert und das Ergebnis bekommt dann ein negatives Vorzeichen. Sind beide Operanden negativ, werden auch die Beträge dividiert, und das Ergebnis ist positiv. Das „Abrunden" bezieht sich hier also auf den Betrag des Ergebnisses.

Und dann wäre da noch der Operator `"%"`. Er liefert den bei der ganzzahligen Division auftretenden Rest. `17/4` liefert also den Quotienten `4,17%4` den Rest `1`. Hier gelten die gleichen Vorzeichenregeln wie bei der Division, was mathematisch nicht gerade schön ist: Nach der mathematischen Definition ist „a modulo b" stets eine Zahl zwischen 0 und b-1, z. B. gilt

```
-17 modulo 4 = 3
(-17/4 = -4.25, abgerundet also -5; 4*(-5) ist -20,
der Rest ist dann (-17)-(-20) = 3)
```

In MaxonC++ liefert `"-17 % 4"` dagegen das Ergebnis `-1`, denn der Quotient ist wegen Abrundung des Betrags `-4, 4*(-4)` ergibt `-16`, der Rest ist also `(-17)-(-16) = -1`.

Warum diese unschöne Definition in MaxonC++? Der Grund ist ganz einfach der, daß der Prozessor MC68000, der in den meisten Amiga steckt, so rechnet.

Da wir gerade dabei sind, den Prozessor schlechtzumachen: er kann noch nicht einmal richtig multiplizieren und dividieren! Die eingebauten Multiplikationsbefehle können lediglich zwei 16-Bit-Zahlen multiplizieren, wobei das Ergebnis dann immerhin ein 32-Bit-Wert ist; bei der Division kann er nur einen 32-Bit-Wert durch eine 16-Bit-Zahl teilen, wobei Ergebnis und Rest dann höchstens 16 Bit breit sein dürfen, sonst ist das Ergebnis undefiniert. Z. B. liefert folgendes Programmfragment

```
int i, j;
i = 2, j = 50000;
cout « i*j;
```

normalerweise ein vollkommen falsches Ergebnis, denn 50000 ist schon zu groß für eine vorzeichenbehaftete 16-Bit-Zahl (also ein `"short int"`).

Was tun? Entweder kaufen Sie sich nun umgehend einen 68030-Karte für Ihren Amiga, denn der kann vernünftig multiplizieren. Alternativ können Sie aber auch die Option ***„IntMult 32"*** im Pull-Down-Menü **„Optionen:Compiler"** aktivieren. Dann benutzt MaxonC++ nicht die eingebauten Multiplikations- und Divisionsoperationen des Prozessors, sondern ruft jeweils Bibliotheksroutinen auf, die richtig rechnen.

Das hat allerdings den Haken, daß diese Funktionsaufrufe um einiges länger dauern als die Prozessoroperationen. Also sollten Sie sich überlegen, ob Sie wirklich die genauen Bibliotheksfunktionen brauchen, denn oft hat man es ja mit relativ kleinen Zahlen zu tun, die die 16-Bit-Grenze nicht überschreiten.

Es sei noch angemerkt, daß MaxonC++, genau wie alle anderen namhaften C-Implementierungen, arithmetische Überläufe nicht abfängt. Sie können also unter Umständen unsinnige Ergebnisse erhalten, wenn Sie Ihre Datentypen nicht breit genug wählen. Das gehört leider zum Konzept von C++ (ist aber eher eine ungeliebte Erbschaft von C): man programmiert ohne Netz und doppelten Boden. Die Philosophie ist nun einmal, daß ein C-Programmierer weiß, was er will. Als Trost für die geringere Sicherheit bekommt man aber höhere Geschwindigkeit, denn das laufende Programm muß keine Zeit darauf verschwenden, irgendwelche Fehler abzufangen.

1.4.3.5 Addition und Subtraktion

Zu den Operatoren `"+"` und `"-"` gibt es nicht viel zu sagen: sie liefern die Summe bzw. die Differenz ihrer beiden Operanden. Die Bindung erfolgt naheliegenderweise von links.

1.4.3.6 Shift-Operatoren>>

Die beiden Operatoren `"<<"` und `">>"` kennen Sie schon: Damit werden Ausgaben nach `"cout"` geschrieben bzw. Eingaben aus `"cin"` geholt. Aber das ist nicht ihre ursprüngliche Funktion: C++ bietet nämlich die Möglichkeit, Operatoren auf Klassen zu überladen, und in `<stream.h>` hat man nun eben zwei Operatoren auf den Klassen `"ostream"` bzw. `"istream"` überladen, um Objekte dieser Klassen - nämlich `"cout"` und `"cin"` - mit anderen Ausdrücken zu verknüpfen und so eine

flexible Schreibweise für Ein- und Ausgaben zu erhalten. Daß man gerade die Operatoren "«" und "»" dafür gewählt hat, hat keine tiefsinnigere Bedeutung.

In Wirklichkeit haben "«" und "»" nämlich eine ganz andere Aufgabe: Sie shiften Bits nach links bzw. nach rechts.

Beispiel:

```
int i, j, k;
i = 13, j = i « 2;
cout « j;
```

Die Zahl 13 hat die Binärdarstellung 1101. Im Ausdruck "i « 2" werden alle Bits um zwei Positionen nach links geschoben und rechts wird mit Nullen aufgefüllt, was 110100 ergibt - und das ist dezimal 52 und somit die Ausgabe unseres Programmfragments. Ein n-facher Linksshift entspricht einer Multiplikation mit 2^n.

Der Rechtsshift-Operator "»" arbeitet analog, nur werden die Bits eben nach rechts verschoben, was einer Division durch 2^n entspricht. Bei vorzeichenbehafteten Datentypen bleibt das Vorzeichen erhalten. Übrigens binden auch diese Operatoren von links nach rechts.

Shift-Operationen braucht man vor allem für maschinennahe Bit-Manipulationen. Es ist ein beliebter Trick, Multiplikationen und Divisionen durch Verschiebungen zu ersetzen, denn mit einem Shift kann man erheblich Rechenzeit sparen. MaxonC^{++} optimiert aber schon selbstständig "8*i" zu "i«3", so daß Sie sich solche Bitfummeleien sparen können.

Wenn sie auch vollkommen unterschiedliche Bedeutungen haben, so haben die beiden bisher beschriebenen Bedeutungen von "«" doch den gleichen Auswertungsvorrang. Z. B. wird

```
cout « i « j;
```

ausgewertet wie

```
(cout « i) « j;
```

d. h. es wird erst "i" und dann "j" ausgegeben. Wenn Sie etwas anderes wollen, müssen Sie

```
cout « (i « j);
```

schreiben. Entsprechend ist

```
cout « i » j;
```

ein Fehler, denn hier werden zuerst "cout" und "i" verknüpft und und dann das Ergebnis und "j" mit "»" verknüpft. Da der Operator "»" aber auf dem Typen von "cout", nämlich der Class "ostream", gar nicht definiert ist, meldet der Compiler hier einen Fehler. Hier müssen Sie also auf jeden Fall Klammern setzen.

Dieser scheinbar unpraktische Sachverhalt kommt daher, daß man in C++ zwar Operatoren überladen und umdefinieren kann, so wie es mit `"<<"` auf `"ostream"` getan wird, aber nicht ihre Auswertungsreihenfolge verändern kann.

1.4.3.7 Vergleiche

Es gibt hier zwei Gruppen von Vergleichen mit unterschiedlichem Vorrang. Da wären zunächst die Vergleiche „kleiner" `( < )`, „kleiner oder gleich" `( <= )`, „größer" `( > )` und „größer oder gleich" `( => )`. Sie vergleichen zwei Zahlenwerte und liefern eine `1`, wenn die Bedingung wahr ist, und sonst eine `0`. Die Bindung geschieht von links nach rechts, aber Vorsicht:

```
0 < n < 1000
```

wird ausgewertet als

```
(0 < n) < 1000
```

Der Ausdruck in den Klammern liefert `0` oder `1`, was stets kleiner als 1000 ist. Also sind obige Ausdrücke immer wahr! Ob eine Zahl wirklich zwischen 0 und 1000 liegt, müßte man mit

```
(0 < n) && (n < 1000)
```

abfragen - den Operator `"&&"` lernen wir weiter unten kennen.

Dann gibt es die Vergleichsoperatoren `==` („gleich") und `!=` („ungleich"). Sie werden nach den anderen Vergleichsoperatoren ausgewertet:

```
a < b == c > d != 1
```

entspricht

```
((a < b) == (c > d)) != 1,
```

was logisch äquivalent zu

```
!((a < b) == (c > d))
```

ist (Sie erinnern sich: `"!"` war die logische Negation) und somit

```
(a < b) != (c > d)
```

entspricht. Das Ganze ist dann so etwas wie ein „ausschließliches Oder", denn der Ausdruck ist wahr, wenn einer der beiden Teilausdrücke - aber nicht beide - wahr ist. Und damit hätte ich auch schon elegant zu den diversen logischen Operatoren übergeleitet.

Achtung: C-Anfänger haben oft ihre Probleme mit dem Vergleichsoperator `==`, denn viele andere Programmiersprachen verwenden hier das einfache `"="`. Fehler wie `"n = 5"` statt `"n == 5"` kommen anfangs oft vor und sind dann schwer zu finden. Aber ich kann Sie beruhigen: Man gewöhnt sich daran.

1.4.3.8 Bitweise Verknüpfungen

Neben dem bereits erwähnten unären bitweisen „Nicht"-Operator `"~"` bietet C++ noch die Bit-Verknüpfungen `"&"` (UND), `"^"` (AUSSCHLIESSLICHES ODER) und `"|"` (ODER). Von diesen hat `"&"` die höchste und `"|"` die niedrigste Auswertungspriorität, ansonsten erfolgt die Auswertung von links nach rechts. Die ganzzahligen Operanden werden Bit für Bit logisch verknüpft. Dabei werden alle Operanden ausgewertet, auch wenn das manchmal nicht nötig wäre: Bei Ausdrücken wie `"0 & x"` ist das Ergebnis ohnehin `0`, so daß der Teilausdruck `"x"` eigentlich nicht weiter betrachtet werden müßte. Dies geschieht aber trotzdem, anders als bei den nun folgenden logischen Verknüpfungen:

1.4.3.9 Logische Operatoren

In C steht die Zahl `0` für `"falsch"` und jede andere Zahl für `"wahr"`. Es gibt also keinen speziellen booleschen Datentypen wie in Pascal. Das ist oft ganz praktisch, denn statt `"i != 0"` kann man deshalb ganz einfach `"i"` schreiben, oder auch `"!i"` statt `"i == 0"`.

Logische Aussagen können mit „Und" sowie „Oder" verknüpft werden. Die Operatoren dazu heißen `"&&"` bzw. `"||"` und liefern als Ergebnis stets `0` oder `1`. Dabei wird die Auswertung ggf. angekürzt: ist bei `"&&"` der linke Operand `0`, also logisch falsch, ist das Ergebnis sowieso `0` und der rechte Operand wird überhaupt nicht mehr betrachtet. Bei `"||"` entfällt entsprechend die Auswertung des rechten Operanden, wenn der linke bereits logisch wahr, also von `0` verschieden, ist.

Ein etwas obskures Beispiel:

```
int i, j;

j = i>=0 && i++;
```

Welchen Wert erhält hier `"j"`, und was geschieht mit `"i"`?

Wenn die linke Bedingung `"i>=0"` falsch ist, ist das Ergebnis `0` und die Auswertung wird abgebrochen, der Wert von `"i"` wird also nicht verändert. Falls `"i"` aber doch positiv ist, muß die rechte Bedingung doch noch ausgewertet werden. Also wird dann `"i"` um 1 erhöht und dann der alte Wert von `"i"` betrachtet. Ist er logisch falsch, also `0`, so erhält `"j"` ebenfalls den Wert `0`, andernfalls wird `1` zugewiesen.

Das Beispiel entspricht also:

```
if (i>=0)
{ j = (i != 0);
     i++;
}
else
   j = 0;
```

was natürlich wesentlich anschaulicher sind. Genaueres über `"if"` und `"else"` erfahren Sie in einem der folgenden Abschnitte.

Der Operator `"&&"` wird vor `"||"` zusammengafaßt. Ansonsten erfolgt die Auswertung logischer Verknüpfungen von links nach rechts.

1.4.3.10 Der bedingte Ausdruck

Wir haben bei den logischen Operatoren schon eine seltsame Möglichkeit kennengelernt, Fallunterscheidungen in Ausdrücken unterzubringen. Viel eleganter geht das mit dem Operator `"?:"`. Dies ist in C++ der einzige Operator, der gleich drei Operanden besitzt: Links vom Fragezeichen steht eine Bedingung. Ist diese Bedingung logisch wahr, so wird der Ausdruck zwischen `"?"` und `":"` ausgewertet, andernfalls ist das Ergebnis der Ausdruck hinter dem Doppelpunkt.

Ein Beispiel:

```
sign = i>0 ? +1 : (i<0 ? -1 : 0);
```

Wie der Name schon andeutet, wird hier das Vorzeichen von `"i"` berechnet. Gilt `"i>0"`, so wird nur `"+1"` ausgewertet, was dann das Ergebnis ist, andernfalls wird ausschließlich der Ausdruck hinter dem ersten `":"` ausgewertet, der ebenfalls ein bedingter Ausdruck ist. Dort wird dann noch auf `"i<0"` getestet und das Ergebnis ist dann entweder `-1` oder `0`.

Im allgemeinen sollte man bedingte Ausdrücke vermeiden und statt dessen Fallunterscheidungen mit `"if"` und `"else"` benutzen:

```
if (i>0)
  sign = +1;
else
  if (i<0)
    sign = -1;
  else
    sign = 0;
```

Dies wird allgemein - zu Recht - als übersichtlicher angesehen.

Da wir gerade bei unübersichtlichen Ausdrücken sind: Die Bindung erfolgt hier von rechts, so daß der Ausdruck hinter dem `":"` ebenfalls ein bedingter Ausdruck sein kann, ohne daß man Klammern setzen muß!

Unser Beispiel könnte man (sollte man aber nicht) also auch so schreiben:

```
sign = i>0 ? +1 : i<0 ? -1 : 0;
```

Blicken Sie da noch auf Anhieb durch? Also, ich nicht! Also Pfoten weg von solchen Dingern!

Das Ganze wird noch chaotischer, wenn hinzter dem Fragezeichen beide Operatoren L-Werte vom gleichen Typ sind. Dann ist nämlich das Ergebnis des Ausdrucks ebenfalls ein L-Wert, dessen Wert man verändern kann:

```
int i, j;
i = 13, j = 14;

++(i>j ? i : j);
```

Hier wird der Operator `"++"` entweder auf `"i"` oder `"j"` angewendet, anhängig von der Bedingung `"i>j"`. Es wird somit die größere Variable inkrementiert; in diesem Fall ist das `"j"`.

Wie Sie sehen, tragen bedingte Ausdrücke nicht gerade zur Lesbarkeit von Programmen bei. Es gibt aber durchaus Situationen, in denen normale Fallunterscheidungen mit `"if"` und `"else"` den Code unangemessen aufblähen würden, so daß man bedingte Ausdrücke manchmal nur schlecht vermeiden kann.

1.4.3.11 Zuweisungen

Den Zuweisungsoperator `"="` kennen Sie ja bereits. Der linke Operand muß ein L-Wert sein, der rechte ein Wert, der in den Typen des linken umgewandelt werden kann. Da wir bisher nur ganzzahlige Datentypen kennen, lassen sich Typen vorerst grundsätzlich ineinander umwandeln. Dabei kann es zu Überläufen kommen, wenn etwa ein zu gorßer Wert an einen schmalen Datentypen zugewiesen wird, was C aber größzügig ignoriert.

Dann gibt es noch die Zuweisungen mit Operator:

```
+=   -=   *=   %=   «=   »=   &=   |=   ^=
```

Jeder Ausdruck

```
A1 op= A2
```

ist eine Kurzschreibweise für

```
A1 = (A1) op (A2),
```

wobei `"A1"` aber nur einmal ausgewertet wird.

```
"n += 1" ist also identisch mit "++n".
```

Man beachte, daß `"i *= 2 + 3"` keineswegs `"i = i * 2 + 3"`, sondern `"i = i* (2+3)"` bedeutet.

Jeder Operator hat in C ein Ergebnis, also auch eine Zuweisung. Ihr Ergebnis ist der Wert des linken Operanden nach der Zuweisung, was dann aber kein L-Wert ist.

So ist

```
a = b = c;
```

identisch mit

```
a = (b = c),
```

was eine beliebte Aukürzung für

```
b = c; a = b;
```

Die Bindung erfolgt hier offensichtlich von rechts nach links.

1.4.3.12 Der Komma-Operator

Die Benutzung des Kommas als Operator ist, ähnlich wie die bedingten Ausdrücke, ein C-Feature, daß manchmal einige Tipparbeit erspart, die Lesbarkeit von Programmen aber alles andere als verbessert. Trotzdem sei dieser Operator der Vollständigkeit halber erwähnt.

Es ist möglich, zwei oder mehr Ausdrücke mit `" , "` aneinanderzuhängen. Sie werden der Reihe nach bewertet - es ist garantiert, daß die Auswertung von links nach rechts erfolgt, was z. B. bei `"+"` nicht versprochen wird - und das Ergebnis ist dann der Wert des letzten Ausdrucks.

Ein Beispiel:

```
int i, j;

j = (i=4, i++, i*i);
```

Hier wird zuerst der Variablen `"i"` der Wert 4 zugewiesen. Anschließend wird ihr Wert um 1 erhöht und zu guter Letzt das Quadrat von `"i"` berechnet. Dieser Wert, also 25, wird dann an `"j"` zugewiesen.

1.4.4 Fließkommatypen

1.4.4.1 Literale und Typen

Bisher haben wir uns nur mit ganzen Zahlen befaßt aber es gibt in C natürlich auch Fließkommazahlen, sogar drei Stück. Sie heißen `"float"`, `"double"` und `"long double"`. MaxonC++ benutzt nur zwei verschiedene Zahlenformate, so daß `"double"` und `"long double"` praktisch identisch sind. Formal gelten sie aber trotzdem als zwei verschiedene Datentypen.

Eine Fließkommakonstante besteht aus einem Vorkommateil, einem Komma, das aber in Wirklichkeit ein Punkt ist, einem Nachkommateil und optinal dem Exponentenzeichen `"E"` mit einem ganzzahligen Exponenten mit oder ohne Vorzeichen. Folgendes sind gültige Literale für Fließkomma- bzw. Gleitpunktzahlen:

```
17.4
26731.
.42
1.2e+3
1E10
0.05e-6
```

Wie Sie sehen, dürfen der Vorkomma- oder der Nachkommateil fehlen, aber natürlich nicht beide (welchen Wert hätte denn auch die Zahl `"."`?) Wird ein Exponent angegeben, darf auch der Dezimalpunkt weggelassen werden. Für Anfänger sei noch erklärt, daß `"1.2E+3"` die auf Computern übliche Schreibweise für `"1.2*10^3"` ist und damit `"1200.0"` entspricht.

Eine solche Fließkommakonstante hat aus unerfindlichen Gründen automatisch den Typ `"double"`. Durch Anhängen von `"f"` oder `"F"`, z. B. `"4.2f"`, wird daraus eine `"float"`-Kon-

stante. Entsprechend macht man sich mit dem Suffix `"l"` oder `"L"` eine `"long double"`-Konstante.

1.4.4.2 Interne Darstellung der Daten

Was sind überhaupt die Unterschiede zwischen den beiden (bzw. drei) Fließkommatypen? Und was sind überhaupt „Fließkommatypen"?

Eine Fließkomma- oder Gleitpunktzahl besteht aus Vorzeichen, Mantisse und Exponent. Ihr Wert ist dann

```
Vorzeichen * Mantisse * Basis^Exponent
```

Als „Basis" wählt man naheliegenderweise die Zahl 2, die Mantisse wird stets auf ein bestimmtes Format „normiert". Z. B. ist die Zahl `6` gerade `0.75 * 8`, also `0.75 * 2^3`. Die Mantisse ist hier also `0.75`, der Exponent `+3`, und das Vorzeichen natürlich positiv. Jede Zahl läßt sich theoretisch in der Form `0.xxxxx * 2^y` darstellen - praktisch stehen für die Mantisse aber nur endlich viele (Binär-)Ziffern zur Verfügung. Deshalb lassen sich Zahlen in der Regel nur als NÄHERUNG darstellen. Schon eine scheinbar so einfache Zahl wie `0.1` hat keine endliche Fließkommadarstellung, da die Mantisse ja binär und nicht dezimal dargestellt werden muß.

Ein `"float"` ist vier Bytes, also 32 Bit, breit. Davon sind 24 Bits Mantisse, 1 Bit steht für das Vorzeichen und 7 Bits für den Exponenten. Das reicht für eine Genauigkeit von ca. 6 Stellen im Bereich von 10 ^ -18 bis 10 ^ +18. Überschreitet eine Rechnung diesen Bereich, tritt ein Überlauf auf, der aber keine Folgen hat, sondern „nur" ein völlig falsches Ergebnis liefert. Alle Zahlen unter etwa 10 ^ -18, oder „1E-18", wie Computer zu sagen pflegen, betrachtet die Amiga-Arithmetik als Null, für die es eine Sonderdarstellung gibt (die Mantisse wird dabei nicht normiert, sondern einfach auf Null gesetzt).

`"float"` ist zwar schon ganz nett, aber für viele Berechnungen reicht die Genauigkeit oder der Exponentenbereich einfach nicht aus. Deshalb gibt es `"double"` bzw. `"long double"`: Diese Datentypen sind 8 Bytes bzw. 64 Bit breit. Sie setzen sich zusammen aus 52 Mantissenbits, 11 Bits für den Exponenten und natürlich wieder einem Vorzeichenbit. Die Genauigkeit der Mantisse beträgt dann fast 16 Dezimalstellen („fast", weil es in der 16ten Ziffer zu Ungenauigkeiten kommen kann) und es sind dezimale Exponenten von +308 bis -308 möglich. Die größte intern darstellbare `"double"`-Zahl ist etwa `1.79769313486232e+308`. Wenn Sie diese Zahl allerdings ausgeben wollen, kommt es dabei in den Bibliotheksfunktionen von MaxonC++ zu Überläufen, deshalb können nur maximal halb so große Zahlen mit `"cout"` ausgegeben werden. Der entsprechende Effekt tritt auch auf, wenn Sie sehr kleine Zahlen im Bereich von 1E-308 ausgeben wollen. Aber diese Einschränkungen sind wohl eher akademisch, denn in der Praxis sollte man es ohnehin vermeiden, bis an den äußersten Rand eines Wertebereichs zu gehen.

MaxonC++ benutzt für die Arithmetik die eingebauten Fließkommaroutinen des Amiga-Betriebssystems. Die `"double"`-Funktionen haben dabei die unangenehme Eigenart, daß bei einem Überlauf oder einer Division durch 0 eine Prozessor-Exception ausgelöst wird. Die integrierte Umgebung von

MaxonC++ fängt solche Exceptions zwar ab, aber wenn ein solches Programm autonom läuft, hängt sich der Task auf. Also sollten Sie in der Tat vorsichtig mit `"double"`-Operationen umgehen.

1.4.4.2 Fließkommaoperationen

Alle artihmetischen Operatoren, die Sie von den ganzzahligen Typen her kennen, arbeiten, soweit das sinnvoll ist, auch auf Fließkommadaten. Der Modulo-Operator `"%"` ist nicht zulässig, denn bei einer Fließkommadivision tritt naturgemäß kein Rest auf. Ebenso sind die Operatoren `"«"` und `"»"` mit einem Fließkommaausdruck als linkem Operanden verboten, denn `"double"` und `"float"` haben ja intern ein etwas kompilziertes Format, und es würde keinen Sinn machen, irgendwelche Mantissenbits in den Exponenten hineinzuschieben. Auch die bitweisen Verknüpfungen `"&"`, `"|"` und `"^"` sind aus demselben Grund für Fließkommaoperanden gesperrt. Folglich sind auch `"%="`, `"«="`, `"»="`, `"&="`, `"|="` und `"^="` nicht erlaubt.

Ein Fließkommaausdruck kann auch als „Bedingung" in einer bedingten Anweisung auftreten. Der Ausdruck ist dann „wahr", wenn er nicht 0.0 ist. Ebenso kann ein Fließkommaausdruck auch Operand der logischen Operatoren `"||"` und `"&&"` sein. Das Ergebnis der logischen Operatoren ist dann aber trotzdem `"int"`.

1.4.5 Auswertung von Ausdrücken

Spätestens jetzt, wo wir zwei grundsätzlich verschiedene Gruppen von Datentypen (nämlich ganzzahlige und Fließkommatypen) kennen, kann ich Ihnen die genauen Auswertungsregeln nicht ersparen. Vor allem werden Sie bestimmt wissen wollen, was geschieht, wenn Operatoren auf Operanden unterschiedlicher Typen arbeiten.

Betrachten wir einmal einen Ausdruck wie `"x + y"`. Hier gelten dieselben Regeln wie bei den meisten anderen binären Operatoren:

- Ist ein Operand `"long double"`, so wird der andere in `"long double"` umgewandelt. Der Operator wird dann in `"long double"` ausgewertet und das ist auch der Typ des Ergebnisses; andernfalls:
- Ist ein Operand vom Typ `"double"`, so wird der Operator entsprechend in `"double"` ausgewertet, sonst:
- Gilt dasselbe analog für `"float"`.

Also wird bei `"17.0 + 4"`, wo der linke Operand `"double"` ist, die 4 ebenfalls nach `"double"` konvertiert, und das Ergebnis ist `"double"`.

Ist aber keiner der beiden Operanden ein Fließkommaausdruck, so gelten die Ganzzahl-Regeln:

Zuerst einmal werden Operanden der `"char"`- und `"short"`-Typen (gleich ob vorzeichenbehaftet oder vorzeichenlos) in `"int"` umgewandelt. Der Compiler lehnt es nun einmal ab, sich mit Zahlen zu befassen, die weniger als 32 Bit breit sind.

- Wenn nun einer der Operanden `"unsigned long"` ist, so wird der Ausdruck in `"unsigned long"` ausgewertet; wenn nicht:
- Ist ansonsten einer der Operanden `"long"`, so erfolgt die Auswertung in `"long"`, andernfalls:
- Falls ein Operand `"unsigned int"` ist, wird auch in eben diesem Datentyp gerechnet.

Wenn alle diese Bedingungen nicht zutreffen, wird der Ausdruck in `"int"` berechnet.

Ein beliebter Fehler ist, `"1/2"` zu schreiben. Wenn Sie sich die obigen Regeln einmal ansehen, werden Sie feststellen, daß hier in `"int"` gerechnet wird. Da aber bei der Integerdivision grundsätzlich das Ergebnis auch `"int"` ist und dementsprechend abgerundet wird, ist `"1/2"` nichts anderes als `"0"`.

Folgende Schreibweisen würden hingegen zum gewünschten Ergebnis, nämlich dem `"double"`-Wert `"0.5"`, führen:

```
1.0/2
1/2.0
1.0/2.0
```

Die hier beschriebenen Umwandlungsregeln gelten für die meisten binären Operatoren. Dazu gehören nicht die logischen Operatoren `"||"` und `"&&"`, denn deren Ergebnis ist immer `"int"` (genaugenommen 0 oder 1) und eine Umwandlung der Operanden erfolgt nicht, denn es kommt ja gar nicht auf deren Wert an, sondern nur darauf, ob ihr Wert Null ist oder nicht.

Bei den Vergleichsoperatoren werden zwar die Operanden genau wie oben beschrieben auf einen einheitlichen Typen gebracht, aber das Ergebnis ist auch hier stets ein logischer `"int"`-Wert.

Dabei kann es durchaus zu Überraschungen kommen, wenn vorzeichenlose und vorzeichenbehaftete Datentypen gleicher Größe verglichen werden:

```
int i;
unsigned int u;

i = -1;
u = +1;

cout « (i > u);
```

Verblüffenderweise ist das Ergebnis hier `"1"`, d. h. `"-1"` ist größer als `"+1"`. Der Grund ist ganz einfach der, daß hier im vorzeichenlosen Bereich verglichen wird. Die `"int"`-Zahl `"-1"` wird also un `"unsigned"` umgewandelt (was ja eigentlich gar nicht geht) und entspricht dort `"0xffffffff"`, also der größten in `"unsigned"` überhaupt darstellbaren Zahl. Grund für diese komische Umwandlung ist die sog. 2-Komplement-Darstellung für negative Zahlen, wie sie heute auf praktisch allen Rechnern angewandt wird.

Der Vollständigkeit halber sei noch erwähnt, daß bei den Zuweisungsoperatoren selbstverständlich die obigen Regeln nicht gelten. Bei der Zuweisung `"="` wird der rechte Operand in den Typen des

linken umgewandelt. Dabei dürfen C-Compiler eine Warnung ausgeben, wenn ein Fließkommawert an eine ganzzahlige Variable zugewiesen wird.

Bei den zusammengesetzten Operatoren wie `"+="` oder `"*="` gilt dann wieder, daß `"a op= b"` absolut identisch mit `"a = a op b"` ist. So ist

```
int i;

i *= 0.5;
```

durchaus (bedingt) sinnvoll: es wird `"i * 0.5"` wie oben beschrieben in `"double"` ausgewertet, das Ergebnis wieder nach `"int"` gewandelt und an `"i"` zugewiesen.

Da aber so gut wie jede Typumwandlung Rechenzeit kostet (Ausnahme: Umwandlung zwischen Ganzzahltypen gleicher Breite), ist dies auch ein Beispiel für schlechte Programmierung, denn

```
i /= 2;
```

oder (bei manchen Compilern sogar noch besser)

```
i >>= 1;
```

würde das gleiche Ergebnis viel schneller liefern.

1.4.6 Explizite Typumwandlungen

Da wir gerade dabei sind, mit Datentypen zu jonglieren, möchte ich noch kurz ein Thema anreißen, das C berüchtigt gemacht hat, aber so schlimm nun auch wieder nicht ist: die expliziten Typkonvertierungen, auch „casting" genannt.

Sie erinnern sich wahrscheinlich noch an das Beispiel, bei dem wir den Compiler gezwungen haben, `"1/2"` als Fließkommaberechnung auszuwerten: es genügte, eine der beiden Konstanten als `"double"`-Literal zu schreiben, also beispielsweise `"1.0/2"`. Aber was geschieht nun, wenn wir es mit Variablen statt Konstanten zu tun haben, etwa so:

```
int i, j;
double d;

i = 1;
j = 2;

d = i/j;
```

Der Ausdruck `"i/j"` wird streng nach den C-Typregeln erst als `"int"` ausgewertet, was also wieder `"0"` ergeben würde, und erst dann wird der Quotient nach `"double"` gewandelt, was offensichtlich nicht erwünscht ist.

Natürlich könnte man die C-Sprachdefinition ändern, etwa so, daß Ausdrücke, die an `"double"` zugewiesen werden, auch in `"double"` ausgewertet werden. Das erscheint auf kurze Sicht als eine

sinnvolle und praxisgerechte Definition. Aber die Typregeln für Operatoren mit zwei Operanden sind schon kompiziert genug, und wenn man diese Regeln dann auch noch vom „gewünschten" Ergebnistyp abhängig macht, werden sie nur noch komplizierter, und dem Programmierer wird das Leben keinesfalls erleichtert.

Also zurück zu unserem Problem. Wir lösen es im Prinzip genau wie beim `"1/2"`-Beispiel: wir sagen dem Compiler, daß (wenigstens) einer der Operanden in ein `"double"` verwandelt wird. Und genau dazu dienen in C die „casts".

Die Sytnax ist zunächst denkbar einfach:

Ein Cast sieht aus wie „(typ) ausdruck", d. h. man schreibt erst den Ziel-Datentyp in runde Klammern und setzt dann den entsprechenden Teilausdruck dahinter.

Also sieht unsere Division folgendermaßen aus:

```
d = (double) i / j;
```

Ein Cast bindet so stark wie ein unärer Operator, aber von rechts nach links. Also ist

```
(double)i++ + j
```

identisch mit

```
( (double) (i++)) + j
```

Auch hier kann man natürlich mit Klammern eingreifen, z. B.

```
(int)(i+j)
```

In C++ gibt es eine alternative Schreibweise für Typwandlungen, die allgemein als lesbarer angesehen wird:

```
int(x)
```

Bei dieser Schreibweise wird also der Typname ohne Klammern geschrieben, während der umzuwandelnde Ausdruck geklammert wird. Bei einfachen Typnamen, wie sie bisher nur besprochen wurden, ist dies eine feine Sache. Wir werden aber bald Beschreibungen für abgeleitete Datentypen kennenlernen, und dann kommt man nicht mehr um die alte Cast-Form herum.

1.4.7 Initialisierungen

Oft hat man, wenn man eine Variable deklariert, schon einen passenden Anfangswert dafür zur Hand, was dann etwa so aussehen könnte:

```
int i, j;
i = 26, j = 731;
```

Weil diese Situation so oft vorkommt, gibt es in C die Möglichkeit der „Initialisierung" von Variablen. Das heißt ganz einfach, daß bei der Variablendeklaration ein Startwert gleich zugewiesen wird, etwa so:

```
int i = 26, j = 731;
```

Für Initialisierungen gelten in C die gleichen Regeln wie bei einer normalen Wertzuweisung. C++ hat einige Features auf Lager, wobei das nicht so ist, aber das werden wir noch beizeiten sehen.

Ein weiterer Unterschied zwischen C und C++: In C müssen in einem Block, also z. B. einer Funktion, immer zuerst die Variablendeklarationen und dann die Anweisungen stehen. In C++ darf man beides nach Belieben mischen:

```
void main()
{
   cout « "Bitte eine Zahl eingeben: ";  // Anweisung
   int i;              // Deklaration
   cin » i;          // Anweisung
   int j = i*i;   // Deklaration
   cout « "Das Quadrat davon ist " « j;  // Anweisung
}
```

Der angenehme Effekt dabei ist, daß man Variablen erst dann deklarieren muß, wenn man einen sinnvollen Wert hat, um sie zu initialisieren. Das hilft, Fehler zu vermeiden, denn es ist im Beispiel nicht möglich, versehentlich `"j"` zu benutzen, bevor es initialisiert ist - ganz einfach deshalb, weil es da noch gar nicht bekannt ist.

1.5 Anweisungen

1.5.1 Bedingungen und Verzweigungen

Zu den wichtigsten Strukturelementen eines Programms gehören sicher Fallunterscheidungen. Im Prinzip gehören bedingte Ausdrücke auch dazu: In Abhängigkeit von einem logischen Ausdruck wird entschieden, welcher Teil eines Ausdrucks ausgewertet werden soll.

Die `"if"`-Anweisung dient zu demselben Zweck, aber auf höherer Ebene: Hier wird zwischen ganzen Anweisungen verzweigt. Ein zugegebenermaßen nicht sehr gehaltvolles Beispiel:

```
#include <stream.h>

void main()
{
   cout « "Bitte eine Zahl eingeben: ";
   int Ein, Aus;
   cin » Ein;

   if (Ein % 2 == 0)
   {
      cout « "Die Zahl ist gerade.\n";
```

```
    Aus = Ein/2;
    cout « "Die Hälfte von " « Ein « " ist " « Aus;
  }
  else
    cout « Ein « " ist ungerade.";
}
```

Eine Zahl ist bekanntlich gerade, wenn sie sich ohne Rest durch 2 teilen läßt, d. h. wenn „Zahl % 2" gerade Null ist. Dies ist die Bedingung, nach der im Beispiel verzweigt wird. Wie Sie hier sehen, muß die Bedingung hinter dem Wortsymbol `"if"` immer in runden Klammern stehen. Als Bedingung ist ein beliebiger numerischer Ausdruck (oder auch ein Zeigerausdruck, wie wir später kennenlernen werden) erlaubt. In C gibt es schließlich, wie bereits erwähnt, keine Trennung zwischen logischen und numerischen Ausdrücken.

Nur wenn die Bedingung wahr (d. h. nicht Null) ist, wird die folgende Anweisung hinter dem `"if"` ausgeführt. Es ist möglich, mehrere Anweisungen mit geschweiften Klammern zu einer einzigen zusammenzufassen, wie es in unserem Beispielprogramm geschehen ist.

Hinter den `"if"`-Zweig einer Fallunterscheidung kann - muß man aber nicht - noch einen `"else"`-Teil setzen, der ausgeführt wird, wenn die Bedingung eben nicht wahr ist. Er besteht aus dem Wortsymbol `"else"`, gefolgt von einer Anweisung. Auch hier können natürlich mehrere Einzelanweisungen mit geschweiften Klammern zusammengefaßt werden.

Es gibt also genaugenommen zwei `"if"`-Anweisungen: Wenn ein `"else"`- Zweig angegeben wird, entscheidet die Bedingung darüber, welche der beiden Alternativen ausgeführt wird. Ohne `"else"` hängt von der Bedingung ab, ob der `"if"`-Zweig ausgeführt wird. In allen Fällen wird anschließend hinter der gesamten `"if"`-Anweisung weitergearbeitet.

Da eine `"if"`-Anweisung natürlich selbst eine Anweisung ist, kann man Fallunterscheidungen selbstverständlich auch ineinander verschachteln. Es gibt hier übrigens eine kleine „Zweideutigkeit":

```
if (Bedingung1)
if (Bedingung2)
  IfTeil;
else
  ElseTeil;
```

Zu welchem `"if"` gehört nun das `"else"`? Es wurde festgelegt, daß es zum zweiten `"if"` gehört. Also ist obige Anweisung identisch mit

```
if (Bedingung1)
{
  if (Bedingung2)
    IfTeil;
  else
    ElseTeil;
}
```

Durch entsprechende Klammerung kann man das `"else"` auch an das erste `"if"` binden:

```
if (Bedingung1)
{
  if (Bedingung2)
    IfTeil;
}
else
  ElseTeil;
```

Am Rande sei noch darauf hingewiesen, daß es dem Compiler absolut egal ist, wie und ob Sie Anweisungen einrücken. Der Programmierer tut dies ausschließlich für sich selbst und wer noch so alles den Quelltext lesen soll. Wenn man also schreiben würde:

```
if (Bedingung1)
  if (Bedingung2)
    IfTeil;
else
  ElseTeil;
```

wäre für den menschlichen Leser klar, was gemeint ist: das `"else"` soll zum ersten `"if"` gehören. Dem Compiler ist das vollkommen egal, für ihn gelten auch nur hier seine geradezu bürokratischen Vorschriften, nach denen das `"else"` nun mal zum zweiten `"if"` gehört, und damit basta. Seien Sie also beim Verschachteln von Verzweigungen vorsichtig, im Zweifelsfall sollten Sie immer Klammern setzen, damit unmißverständlich klar wird, was wozu gehört. Das ist also ganz analog zum empfohlenen Setzen zusätzlicher runder Klammern in komplizierten Ausdrücken.

1.5.2 Anweisungen, Deklarationen und Blöcke

Leider wird es wieder einmal Zeit, daß wir uns in Formalitäten ergehen (man könnte fast glauben, das Programmieren sei von Beamten erfunden worden...). Zunächst soll noch einmal präzisiert werden, was eine „Anweisung" ist. Wir kennen nämlich inzwischen vier verschiedene Arten davon:

1. Ausdrucks-Anweisungen: Sie bestehen aus einem Ausdruck, der ausgewertet wird, gefolgt von einem Semikolon. Ein Ausdruck ist im wesentlichen alles das, was in 1.4 beschrieben wurde, nämlich Variablen und Konstanten, die durch Operatoren verknüpft werden. Auch ein trivialer Ausdruck wie „42" ist eine Anweisung, bei der aber nichts geschieht. MaxonC++ erzeugt nicht einmal Code für so etwas. Aber es gibt ja durchaus Operatoren, die Auswirkungen auf ihre Operanden haben oder sonst eine Operation einschließen, etwa der Zuweisungsoperator `"="` oder die Ein- und Ausgabeoperatoren `"«"` und `"»"`.

2. Die `"if"`-Anweisung: evtl. mit `"else"`-Teil. Das Ganze Gebilde gilt dann nachher als eine einzelne Anweisung. Im folgenden werden wir noch viele ähnliche Strukturanweisungen betrachten, insbesondere Schleifenanweisungen.

3. Eine Deklaration gilt in C++ ebenfalls als Anweisung. In C ist das anders: Hier müssen Deklarationen und Anweisungen streng voneinander getrennt werden, nämlich dergestalt, daß in einem „Block" immer zuerst alle Deklarationen und dann die Anweisungen stehen müssen.

4. Eine Block-Anweisung: Bisher haben wir sie lässig damit abgetan, daß mehrere Anweisungen mit geschweiften Klammern "`{ ... }`" zusammengefaßt werden. Das ist aber bei weitem nicht alles, und darum soll dies im folgenden näher unter die Lupe genommen werden.

Zunächst einmal ist der Anweisungsteil einer Funktion, z. B. "`main`", denn andere Funktionen haben wir bis hier noch nicht gehabt, nichts anderes als ein „Block", und formal ist das eine Folge von Anweisungen (die in C++ ja, wie schon wiederholt gesagt, auch Deklarationen einschließen).

Vor allem aber hat ein Block einen eigenen „Gültigkeitsbereich", oder auf neudeutsch „Scope". Das bedeutet: Namen, die innerhalb eines Blocks deklariert werden, insbesondere Variablen - sind außerhalb gar nicht bekannt.

Ein Beispiel aus der Praxis:

```
#include <stream.h>

void main()
{  // Anfang des Haupt-Blocks der Funktion "main"

   cout « "Wollen Sie in i)nt oder d)ouble rechnen? ";
   char i;
   cin » i;

   if (i == 'i')
   {  // Hier beginnt ein Block
      cout « "Ihre Eingabe war: " « i « "\n";
      int i, j, erg;
      cout « "Bitte zwei ganze Zahlen: ";
      cin » i » j;
      erg = i/j;
      cout « "Quotient: " « erg;
   }  // Das ist das Ende eines Blocks

   else
      if (i == 'd')
      {  // Noch ein Block
         cout « "Danke für Ihre Eingabe " « i « ".\n";
         double x, y, erg;
         cout « "Bitte zwei Zahlen: ";
         cin » x » y;
         erg = x/y;
         cout « "Quotient: " « erg;
      }  // Ende dieses Blocks
      else
      {
         cout « "Das war eine falsche Eingabe.";
      }

   cout «    "\nProgrammende - Variable \"i\
               " hat immer noch den Wert " « i;
   // Die einzige Variable, die an dieser Stelle bekannt ist
```

```
} // Funktionsende
```

Bei diesem kleinen Programm hat der Benutzer die Wahl, ob er ganzzahlig oder in Fließkomma dividieren will. Dazu wird zuerst ein Zeichen abgefragt, das dann in der `"char"`-Variablen `"i"` abgelegt wird. Nun folgt eine Verzweigung, deren `"if"`-Teil ein eigener Block ist.

Zunächst wird hier die `"char"`-Variable `"i"` ausgegeben, nur um zu zeigen, daß Bezeichner aus höheren Blöcken durchaus in tiefergelegenen (nicht „tiefergelegten", Manta-Witze sind schließlich total out) Blöcken bekannt sind und benutzt werden dürfen. Aber nun kommt etwas Überraschendes: In der folgenden Deklarationszeile wird unter anderem eine neue `"i"`-Variable definiert, diesmal sogar als `"int"`!

Das gehört zum Konzept der „Gültigkeitsbereiche": Bezeichner dürfen in verschiedenen Bereichen durchaus für ganz verschiedene Zwecke definiert und benutzt werden. In diesem Fall überdeckt das neue `"i"` das alte, so daß wir von jetzt an nur noch das neue `"i"` aus dem inneren Block benutzen können. Außerdem werden zusätzlich noch die Variablen `"j"` und `"erg"` definiert.

Der innere Block endet bei den geschweiften Klammern, und wir befinden uns nun wieder auf der Ebene des äußeren Blocks. Das bedeutet vor allem, daß die „innen" deklarierten Variablen `"i"`, `"j"` und `"erg"` jetzt nicht mehr bekannt sind und nicht mehr benutzt werden können. Genaugenommen existieren diese Variablen überhaupt nicht mehr, ihr Speicherplatz könnte also prinzipiell wiederverwendet werden. Dafür hat `"i"` jetzt wieder die alte Bedeutung, nämlich als `"char"`-Variable mit der Benutzereingabe.

Nun ist da noch der `"else"`-Teil, ebenfalls mit einem eigenen Scope. Hier werden die Variablen `"x"`, `"y"` und `"erg"` dazudefiniert. Das zuvor definierte `"erg"` war ja „lokal" zu einem inneren Block und deshalb auf Ebene der Funktion überhaupt nicht bekannt. Also ist es in diesem neuen Block erst recht nicht bekannt und kann getrost neu definiert werden, diesmal als `"double"`.

Und dann hat auch noch der `"else"`-Teil der zweiten `"if"`-Anweisung einen eigenen Block. Weil dieser nur eine einzige Anweisung enthält und keinerlei neue Bezeichner definiert werden, wäre es überhaupt nicht nötig gewesen, hier einen neuen Block zu definieren.

Die letzte `"cout"`-Zeile gehört wieder zum obersten Block der Funktion. Die einzige Variable, die an dieser Stelle bekannt ist, ist `"i"`. Alle Variablen aus tiefer liegenden Blöcken sind hier nicht mehr definiert. Nebenbei bemerkt wird hier auch demonstriert, wie man doppelte Anführungszeichen in einen String einschließt, indem man einen Backslash davorstellt.

Als Anweisung hinter `"if"` und `"else"` darf keine einzelne Deklaration stehen, z. B.

```
if (Bedingung)
  int i; // Error
else
  int j; // Error
```

Das ist ja auch klar, denn was soll obige Anweisung denn auch wohl bedeuten? Sollen `"i"` und `"j"` nur in Abhängigkeit von einer Bedingung existieren, oder wie oder was oder warum auch nicht? Mit einem Block ist das natürlich egal, etwa

```
if (Bedingung)
   { int i; }
else
   { int j; }
```

Hier sind `"i"` und `"j"` nur in ihren kleinen Blöcken bekannt, so daß die Gültigkeit wieder offensichtlich ist.

Übrigens: In C muß hinter jeder Anweisung ein Semikolon stehen. Die einzige Ausnahme ist ein Block, hier darf man das `" ; "` weglassen. Manchmal, z. B. zwischen `"if"` und `"else"`, muß man es sogar unterschlagen, denn ein überflüssiges Semikolon gilt in der C-Syntax als Leeranweisung, so daß bei

```
if (Beding)
   { Block; };          // letzteres Semikolon überflüssig!
else
   usw;
```

gleich zwei Anweisungen zwischen `"if"` und `"else"` stehen, was ja nicht zulässig ist.

1.5.3 Schleifen

1.5.3.1 Die „while"-Anweisung

Zu den unverzichtbaren Elementen der Programmierung gehören Schleifenstrukturen. Dabei wird allgemein eine Folge von Anweisungen wiederholt ausgeführt, bis eine wie auch immer geartete Endbedingung erfüllt ist.

Die elementarste Schleifenform ist in C die `"while"`-Schleife. Sie hat die Form

```
while (Bedingung) Anweisung;
```

Die „Anweisung" wird solange wiederholt, wie die „Bedingung" wahr (ungleich Null) ist. Möglicherweise ist die Bedingung von Anfang an nicht erfüllt - dann wird der Schleifenrumpf überhaupt nicht ausgeführt.

Ein Beispiel:

```
#include <stream.h>

main()
{
   int i = 1, sum = 0, max = 100;
   while (i <= max)
   { sum += i;
     i++;
   }
```

```
   cout « "Ergebnis: " « sum;
}
```

Die Variable `"i"` wird so lange von `0` hochgezählt, bis ihr Wert `100` überschreitet. Dabei werden die Werte jeweils in `"sum"` aufaddiert. Als Ergebnis wird dann die Summe der ganzen Zahlen von `1` bis `100` ausgegeben, nämlich `5050`.

Echte C-Freaks hätten übrigens die Schleife abgekürzt:

```
while (i <= max)
   sum += i++;
```

Mathematiker wissen natürlich, daß man das Ergebnis auch direkt als

```
sum = max * (max+1) / 2;
```

berechnen könnte.

1.5.3.2 Die „for"-Schleife

Die Zählschleife tritt so häufig auf, daß es dafür eine verkürzte Schreibweise gibt: die `"for"`-Schleife. Sie hat die Form

```
for (Anweisung1; Bedingung; Ausdruck)
   Anweisung2;
```

Die erste Anweisung wird einmal zu Anfang ausgeführt. Dann wird die Schleife so lange ausgeführt, wie die Bedingung erfüllt ist, wobei der `"Ausdruck"` nach jedem Schleifendurchlauf ausgewertet wird.

Sie entspricht also

```
Anweisung1;
   while (Bedingung)
   {
   Anweisung2;
   Ausdruck;
   }
```

Dementsprechend können wir unser kleines Beispiel umschreiben:

```
#include <stream.h>

main()
{
   int i, sum = 0, max = 100;

   for (i=1; i <= max; i++)
      sum += i;

   cout « "Ergebnis: " « sum;
}
```

Im Gegensatz z. B. zu Pascal ist "for" in C ein sehr allgemeines Sprachkonstrukt. Man kann wegen der Verwandtschaft mit der "while"-Schleife weit mehr damit machen, als nur eine Variable rauf- oder runterzuzählen. Die Einsatzmöglichkeiten von "for" sind genauso vielfältig wie die von "while".

Bei "for" sind alle drei Ausdrücke bzw. Anweisungen in den Klammern optional. So ist "for (;;) Anweisung;" identisch mit einer endlosen Schleife (eine fehlende Schleifenbedingung wird als „falsch" betrachtet), und "for(;Bedingung;) Anweisung" entspricht vollkommen der "while"-Schleife.

In C++ darf man eine Schleifenzählvariable auch innerhalb der "for"-Startanweisung deklarieren:

```
for (int i=0; i != 10; --i)
   { BlaBla; }
```

Die Variable gehört dann aber zum Scope, in dem die "for"-Schleife liegt, und nicht zum Gültigkeitsbereich des Schleifenrumpfs. Deshalb dürfen Sie bei einer nachfolgenden Schleife "i" zwar wieder benutzen, aber nicht noch einmal deklarieren.

1.5.3.3 Die „do"-Schleife

Die dritte Schleifenkonstruktion von „C" ist die "do"-Anweisung. Bei "while" und "for" wird die Schleifenbedingung schon vor dem ersten Schleifendurchlauf getestet, und wenn sie von Anfang an „nicht erfüllt" ist, wird die Schleife keinmal durchlaufen. Bei "do" dagegen wird die Bedingung erst am Schleifenende und somit nach der ersten Ausführung des Schleifenrumpfs geprüft. Deshalb wird eine "do"-Schleife immer mindestens einmal ausgeführt.

Die "do"-Anweisung hat die Form:

```
do
   Anweisung;
while (Bedingung);
```

Zwischen "do" und "while" darf auch hier nur eine einzige Anweisung (bzw. ein Block) stehen. Bei der "repeat"-Anweisung in Pascal, die in etwa dem "do-while" von C entspricht, bilden ja "repeat" und "until" eine Klammerung um beliebig viele Anweisungen.

Auch die "do"-Schleife wird durchlaufen, so lange die Bedingung erfüllt ist (die "repeat"-Anweisung in Pascal wird ausgeführt, so lange sie nicht erfüllt ist).

Ein typisches Beispiel für "do" sind Wiederholungen von Benutzerfunktionen, etwa so:

```
char c1;
do
{
   DatenDrucken; // Irgendwelches Zeugs auf den Drucker ausgeben
   cout « "Soll ich die Daten noch einmal ausdrucken? (j/n) ";
   cin » c1;
}
while (c1 == 'j');
```

Im Gegensatz zur `"while"`-Schleife darf die Variable „c1", die in der Schleifenbedingung benutzt wird, vor dem ersten Schleifendurchlauf undefiniert sein, denn die Bedingung wird ja erst nach der ersten Ausführung des Schleifeninhalts getestet.

1.5.4 Sprünge

1.5.4.1 Sinn und Unsinn von „goto"

Schleifenstrukturen sind die wichtigsten Elemente strukturierter Programmierung. Es gibt aber Gelegenheiten, bei denen man es nur schwer umgehen kann, wie in alten FORTRAN- oder BASIC-Zeiten mit der `"goto"`-Anweisung quer durch ein Programm zu springen.

Das Ziel eines `"goto"`-Sprungs wird in C durch ein „Label", das durch einen Bezeichner repräsentiert wird, bestimmt. Eine `"goto"`-Anweisung hat dann die Form

```
goto Label;
```

Die Sprungmarke kann man in der Form

```
Label: IrgendEineAnweisung;
```

vor eine beliebige Anweisung stellen. Dabei bilden dann die Marke und die nachfolgende Anweisung zusammen formal eine einzige Anweisung.

Solche Sprünge ermöglichen dann wirklich ganz scheußliche Programmabläufe. Ein Beispiel:

```
void main()
{
  int i, j=0, s;

  if (EgalWas == IrgendWas)
  {
    i = 0;
    goto Dahin;
  }

  for (i=10; i <= 20; ++i)
  {
    s = i*i;
    Dahin: j += s;
  }
}
```

Die ganze schöne Struktur der `"for"`-Schleife geht hier vor die Hunde: Wenn die obskure Bedingung `"EgalWas == IrgendWas"` erfüllt ist, wird `"i"` mit `0` initialisiert und ohne jede Gnade in die `"for"`-Schleife eingesprungen. Dabei wird auch die Schleifenanfangsanweisung übergangen, so daß jetzt nicht von 10, sondern von 0 aus gezählt wird. Dann benutzen wir zu allem Überfluß auch noch die Variable `"s"`, die gar keinen definierten Wert besitzt, so daß `"j"` hier einen wahrschein-

lich unsinnigen Wert erhält - ein typischer Bug, wie er durch unüberlegte `"goto"`'s oft verursacht wird.

Jeder Bezeichner hat bekanntlich einen Gültigkeitsbereich. Die Sprungmarke `"Dahin"` wird im obigen Programm aber scheinbar in einem Block definiert und in einem ganz anderen benutzt. Das scheint aber wirklich nur so, denn die Labels einer Funktion haben einen eigenen Scope. Sie kollidieren nicht mit anderen Bezeichnern der Funktion - man könnte also die Verwirrung komplett machen und denselben Namen für eine Variable und ein Label wählen - und sind in der gesamten Funktion gleichermaßen bekannt. Diese Verletzung der gewohnten Gültigkeitsregeln stellen einen Grund mehr da, auf Sprünge möglichst zu verzichten.

Immerhin wird bei `"goto"` aber ganz scharf darauf geachtet, daß keine Initialisierung übersprungen wird:

```
goto Dahin;
for (i=10; i <= 20; ++i)
{ int s = i*i;
  Dahin: j += s;// Error: Jump past initialisation of "s"
}
```

Hier wird bei der Deklaration des Labels ein Fehler wie **„Jump past initialisation of object 'i'"** gemeldet, d. h. der Compiler stört sich daran, daß hinter die Initialisierung einer Variablen gesprungen wird. Das kommt daher, daß eine Initialisierung keine normale Wertzuweisung ist: Jedes Datenobjekt kann genau einmal, bei seiner Deklaration, initialisiert werden. Nun wird aber mit dem `"goto"` die Initialisierung von `"i"` übersprungen. Sinn und Zweck einer Initalisierung ist aber, daß Variablen von Anfang an einen „sinnvollen" Wert haben, deshalb wurde es generell verboten, auf irgendeine Art eine Initialisierung zu übergehen. Alternativ könnte man vielleicht festlegen, daß der Compiler solche Deklarationen irgendwie „herauszieht", so daß die Variable auf jeden Fall noch vor dem `"goto"` initialisiert wird. Der dazu nötige Aufwand stünde aber in keinerlei Relation zum Nutzen, und so hat man es beim schlichten Verbot solcher Strukturen belassen.

Es ist aber sinnvollerweise erlaubt, Initialisierungen von Variablen, die beim Sprungziel sowieso nicht mehr bekannt sind, zu überspringen:

```
for (Dieses; und; jenes)
{  i = IrgendEinWert;
   if (i > 1000)
      goto NotAusstieg;
   long TotalEgal = i*i;
}
NotAusstieg:
```

Die Initialisierung von `"TotalEgal"` darf getrost übersprungen werden, da diese Variable bei der Sprungmarke `"NotAusstieg"` nicht bekannt ist und folglich `"total egal"` ist, ob sie nun einen sinnvollen Wert hat oder nicht.

Abschließend ist zu sagen, daß man Sprünge nur verwenden sollte, wenn der normale Programmablauf gestört wird und es deshalb nötig ist, aus einer Struktur auszuspringen.

1.5.4.2 „break" und „continue"

Im letzten Beispiel wurde ein `"goto"` benutzt, um eine Schleife zu verlassen. Das geht in C aber auch eleganter: mit der `"break"`-Anweisung.

Ihre Syntax ist denkbar einfach: `"break;"`.

Diese Anweisung beendet sofort die nächsthöhere `"while"`-, `"for"`-, `"do"`- oder `"switch"`-Anweisung (letztere wird weiter unten behandelt). Die Programmausführung wird dann hinter der jeweiligen Strukturanweisung fortgesetzt, so daß `"break"` einem `"goto"` zu einem imaginären Label hinter der Schleife entspricht.

Unser letztes Beispiel läßt sich eleganter (jedenfalls unter Vermeidung eines `"goto"`) so schreiben:

```
for (Dieses; und; jenes)
{ i = IrgendEinWert;
  if (i > 1000)
    break;
  long TotalEgal = i*i;
}
```

Es gibt tatsächlich Situationen, bei denen man weder am Anfang noch am Ende einer Schleife feststellen kann, ob das Ende der Schleifenausführung erreicht sein soll, und diese Entscheidung vielleicht sogar an MEHREREN Stellen im Schleifenrumpf getroffen werden muß. Dann bietet sich folgende Konstruktion an:

```
for(;;)// eigentlich eine Endlosschleife
{ EinPaarAnweisungen;
  if (IrgendWas)
    break;
  NochEinigeAnweisungen;
  if (WasWeissIch)
    { break };
  UndNochWas;
  // usw..
}
```

Wie Sie beim zweiten `"if"` erkennen, kann `"break"` durchaus aus mehreren Blöcken auf einmal ausspringen. Es bezieht sich eben, wie gesagt, auf die nächsthöhere Schleifen- oder `"switch"`-Anweisung.

Mit `"break"` kann man die meisten `"goto"` auf halbwegs strukturierte Weise ersetzen. Und es gibt noch eine weitere Anweisung, mit der man Sprünge vermeiden kann: `"continue"`.

Auch diese Anweisung hat eine simple Syntax, nämlich `"continue;"`. Sie bezieht sich auf die nächsthöhere Schleifenanweisung und setzt die Programmausführung am Anfang des Schleifenrumpfs fort, überspringt also den gesamten Rest des Rumpfs. Bei

```
for(Start; Bedingung; Ausdruck)
{
   Anweisung1;

   continue;

   Anweisung2;

   HierKoennteEinLabelStehen:
}
```

wird nach dem `"continue"` der `"Ausdruck"` ausgewertet, also z. B. die Schleifenzählvariable inkrementiert, die `"Bedingung"` geprüft und dementsprechend die Schleife fortgesetzt oder auch nicht. Ergo entspricht `"continue"` hier einem `"goto HierKoennteEinLabelStehen"`.

Bei `"while"` sieht das Ganze ähnlich aus:

```
while (Bedingung)
{
   Anweisung1;

   continue;

   Anweisung2;

   // Ende des Schleifenrumpfs
}
```

`"continue"` übergeht `"Anweisung2"` und geht direkt ans Ende des Schleifenrumpfs bzw. zu der Stelle, an der die `"Bedingung"` getestet wird.

Auch bei `"do"` ist die Bedeutung von `"continue"` einleuchtend:

```
do
{
   Anweisung1;

   continue;

   Anweisung2;

   // Ende des Schleifenrumpfs
} while (Bedingung);
```

Auch hier wird zu der Stelle gesprungen, an der die Schleifenbedingung ausgewertet wird.

Mit `"break"` und `"continue"` dürfte es möglich sein, die meisten `"goto"`-Anweisungen zu ersetzen. Leider gibt es kein „sauberes" Sprachkonstrukt, das die zweit- oder dritthöhere Schleife verläßt, so daß man dann doch auf ein `"goto"` zurückgreifen muß.

1.5.5 Vielfachverzweigungen

Bisweilen muß man in Abhängigkeit von einem Wert verzweigen, und das nicht, wie bei einem einfachen `"if"`, zwischen zwei, sondern zwischen etlichen Programmstellen. Die `"switch"`-Anweisung stellt eine bequeme und schnelle Abkürzung für Strukturen der Art

```
if (val == Wert1)
   { ... }
else if (val == Wert2)
   { ... }
else if (val == Wert3)
   { ... }
else if (val == Wert4)
   { ... }
else ...
```

dar. Dabei muß `"val"` allerdings ein ganzzahliger Ausdruck sein, und bei `"Wert1"`, `"Wert2"` usw. muß es sich um konstante Ausdrücke handeln, die also bereits vom Compiler und nicht erst zur Laufzeit des Programms ausgewertet werden können.

Eine `"switch"`-Anweisung hat die Form

```
switch (val)
   Anweisung;
```

wobei man als `"Anweisung"` im allgemeinen wohl einen ganzen Block wählen wird. In diese Anweisung setzt man Sprungmarken der Form

```
case Wert:
```

Beim `"switch"` wird dann in Abhängigkeit von `"val"` zum zugehörigen `"case"` verzweigt.

Ein Beispiel:

```
cout « "Daten L)aden, S)peichern, D)rucken oder P)rogrammende: ";
char c;
cin » c;

switch (c)
{
   case 'l': case 'L':
      Laden;
      break;
   case 's': case 'S':
      Speichern;
      break;
   case 'd': case 'D':
      Drucken;
      break;
   case 'p': case 'P':
      cout « "Sind Sie wirklich sicher? "; cin » c;
      if (c != 'j' && c != 'J')
```

```
        break;
      ProgrammEnde;
        break;
  default:
    cout « "Falsche Eingabe!\n";
    break;
}
```

Der Benutzer wird hier aufgefordert, mit Eingabe eines Zeichens aus einem kleinen Menü auszuwählen. Als erste Alternative wird bei Eingabe von `'l'` oder auch `'L'` der Menüpunkt `"Laden"` ausgeführt. Das anschließende `"break"` ist unbedingt nötig, um aus dem `"switch"` auszusteigen, denn sonst würde ganz einfach in der nächsten Zeile weitergemacht, also mit der Programmfunktion `"Speichern"`. Hier wurde die Programmiersprache wohl nicht unbedingt sinnvoll definiert, denn es dürfte so gut wie nie vorkommen, daß man nach einer `"switch"`-Verzweigung ohne `"break"` gleich mit der nächsten Alternative weitermachen will, während man andererseits leicht ein solches `"break"` vergißt und sich dann wundert, was das Programm denn macht. Aber mit dieser Unzulänglichkeit von C wird man wohl leben können und müssen.

Jedenfalls ist das Ganze so gedacht, daß die `"case"`-Angaben lediglich Labels sind, zu denen beim `"switch"` wie mit einem `"goto"` verzweigt wird. Also darf man auch - wie oben geschehen - beliebig viele `"case"` vor eine einzige Anweisung setzen. Andererseits darf jeder Wert nur in einem `"case"` auftreten, z. B. könnte man nicht zweimal `"case 'p'"` verwenden - denn wohin sollte dann auch verzweigt werden?

Das Label `"default"` darf ebenfalls in jedem `"switch"` höchstens einmal auftreten. Hierhin wird verzweigt, wenn der fragliche Wert bei keinem `"case"` angegeben wird. Wenn kein `"default:"` angegeben wird, wird in diesem Fall die `"switch"`-Anweisung ganz übergangen. Übrigens muß `"default"` keineswegs das letzte Label sein.

Das `"break"` hinter der letzten Anweisung eines `"switch"` ist eigentlich überflüssig, denn am Ende wird die `"switch"`-Anweisung ja sowieso verlassen. Es ist aber empfehlenswert, es hier trotzdem zu setzen, denn oft hängt man an ein `"switch"` noch nachträglich Alternativen an, und dann vergißt man leicht das trennende `"break"` - Folgen: siehe oben.

Bei `"switch"` gelten im Zusammenhang mit Initialisierungen genau dieselben strikten Regeln wie bei `"goto"`:

```
switch(i)
{ case 0:
    int j = 1;
  break;
  case 1:
    j = 26731;
    int k = j+1;
}
```

Hier beschwert sich der Compiler, daß bei der Verzweigung nach `"case 1"` die Initialisierung von `"j"` übersprungen wird. Allenfalls in der letzten Alternative darf, wie hier bei `"k"`, eine Initialisie-

rung stattfinden, denn die kann ja nicht mehr übersprungen werden, jedenfalls nicht innerhalb ihres Blocks. Deshalb bietet es sich an, einem `"case"` einen eigenen Block zu spendieren, um Ärger mit Initialisierungen lokaler Variablen zu umgehen:

```
switch(i)
{ case 0:
  { int j = 1;
    break;
  } // Hier ist der Block nötig
  case 1:
    j = 26731;
    int k = j+1;   // Das darf so bleiben
}
```

1.5.6 Die „return"-Anweisung

Das wären dann auch schon (fast) alle Anweisungstypen, die es in C++ gibt. Der Vollständigkeit halber soll hier noch die `"return"`-Anweisung erwähnt werden, über die Sie genaueres im Kapitel 1.6.3 erfahren. Sie beendet die Ausführung einer Funktion und kann optional auch einen Wert zurückgeben. Aber damit gehört sie auch schon in das folgende Kapitel:

1.6 Funktionen

1.6.1 „main" allein genügt nicht

Neben strukturierten Anweisungen - also etwa übersichtlicher Schleifenstrukturen statt wilder Sprünge - ist die Abstraktion durch Funktionsdefinition eine zentrale Idee der strukturierten Programmierung. Die Idee ist, weitgehend in sich abgeschlossene Programmteile zu einer Funktion zusammenzufassen und diese am besten auch noch so allgemein zu formulieren, daß sie möglichst allgemein zu benutzen ist.

Bisher hatten wir es immer mit nur einer Funktion zu tun, nämlich `"main"`. Das ist insofern eine besondere Funktion, als daß der Linker bei der endgültigen „Montage" des Programms darauf besteht, daß es eine solche Funktion gibt, und dann dafür sorgt, daß die Funktion `"main"` beim Programmstart ausgeführt wird. Für den Compiler gibt es dagegen keinen formalen Unterschied zwischen der Funktion `"main"` und einer beliebigen anderen.

Wir können also z. B. eine Funktion `"hello"` definieren, zunächst einmal ohne Rückgabewert (also `"void"`):

```
#include <stream.h>

void hello()
{ cout « "Hello, World!";
}
```

Die leeren Klammern `"()"`, die wir schon bei der Funktion `"main"` stets definiert haben, sind die „Parameterliste" der Funktion `"hello"`. Da diese Liste offensichtlich leer ist, hat die Funktion keine Parameter.

Eine Funktion wird mit Namen und Parameterliste aufgerufen, z. B. so:

```
#include <stream.h>

void hello()
{ cout « "Hello, World!";
}

void main()
{
  hello();       // Funktion "hello" wird ausgeführt
  cout « "\n";   // Zeilenvorschub
  hello();       // Und noch einmal "hello"
}
```

Ungeheuer wichtig: Auch wenn die Funktion gar keine Parameter hat, muß man in C (im Gegensatz z. B. zu Pascal) beim Aufruf eine leere Parameterliste anhängen. Der Funktionsname allein bezeichnet nur die Adresse einer Funktion. Der Postfix-Operator `"()"` (in C gilt eine Parameterliste tatsächlich als Operator) angewendet auf den Namen einer Funktion ruft eben diese auf.

Ein Funktionsaufruf ist übrigens ein Ausdruck und hat deshalb auch einen Datentyp, nämlich jeweils den Ergebnistyp der Funktion. In unserem Fall ist das der Typ `"void"`, so daß der Funktionsaufruf keinen wirklichen Wert hat und man mit dem Ergebnis auch nicht so sonderlich viel machen kann.

1.6.2 Parameter und Argumente

Jetzt sind Sie bestimmt schon neugierig, was es genau mit diesen Klammern `"()"` auf sich hat, denn wenn von einer leeren Parameterliste die Rede war, kann man bestimmt auch etwas da rein schreiben... Und genau davon soll jetzt die Rede sein.

Ein „Parameter" ist ein Wert, der beim Aufruf an eine Funktion übergeben wird. Praktisch geschieht das, indem eine Variable der Funktion „von außen" initialisiert wird. Diese Parametervariablen werden innerhalb der Parameterliste angegeben, z. B. so:

```
void f(int i, int j)
```

Aus historischen Gründen (sprich: wieder eine lästige Erbschaft von veralteten C-Standards) kann man hier nicht wie bei einer gewöhnlichen Variablendefinition

```
void f(int i, j) // Achtung, FALSCH!
```

schreiben. Aber die Anzahl der Parameter ist natürlich beliebig, und es können auch unterschiedliche Datentypen benutzt werden, z. B.

```
void NochEineFunktion(int i, char c, double d);
```

Jedenfalls hat die Funktion `"f"` nun zwei Variablen `"i"` und `"j"`, die beim Aufruf initialisiert werden. Die folgende Funktion berechnet nach dem euklidischen Algorithmus den größten gemeinsamen Teiler seiner Parameter:

```
void ggt(int i, int j)
{
   cout « "Der größte gemeinsame Teiler von " « i « " und "
   « j « " ist: ";

   while (i) // Kurzschreibweise für "while (i != 0)"
   { int hilf = j % i;
     j = i;
     i = hilf;
   }
   cout « j;
}
```

Wie Sie sehen, kann man Parameter genau wie gewöhnliche Variablen benutzen und ihnen sogar einen anderen Wert zuweisen.

Beim Aufruf einer Funktion sind dann geeignete Werte für die Parameter anzugeben, etwa

```
void main()
{ int a, b;
   cout « "Bitte zwei Zahlen eingeben: "; cin » a » b;
   ggt(a,b);
}
```

In C werden normalerweise nur Werte als Parameter übergeben. Die Variablen `"a"` und `"b"` des Hauptprogramms behalten deshalb ihre Werte, auch wenn in der Funktion `"ggt"` den Parametervariablen neue Werte zugewiesen werden.

An dieser Stelle sind dringend einige Anmerkungen zur Sprechweise nötig: Man unterscheidet nämlich zwischen „Parametern" und „Argumenten". In unserem Beispiel sind `"i"` und `"j"` die Parameter der Funktion `"ggt"`, während `"a"` und `"b"`, also die Werte, mit denen die Parameter initialisiert werden, „Argumente" genannt werden. Dementsprechend ist

```
(int i, int j)
```

eine Parameterliste und

```
(a, b)
```

eine Argumentliste. In Pascal ist es dagegen üblich, von „formalen" und „aktuellen" Parametern zu sprechen, also auch von „formalen Parameterlisten" (statt „Parameterliste") bzw. „aktuellen Parameterlisten" (statt „Argumentliste").

Bei der Parameterübergabe gelten dieselben Typregeln wie bei Initialisierungen (das sind bisher noch dieselben wie bei einer gewöhnlichen Wertzuweisung). Also werden auch Datentypen von Argumenten in die der Parameter umgewandelt. Beim Aufruf

```
ggt(17.0, 4.2)
```

würden folglich die Integerwerte „17" und „4" an die Parameter zugewiesen, wobei C-Compiler wieder eine Warnung ausgeben dürfen, was MaxonC++ aber unterläßt.

1.6.3 Rückgabe von Werten

Wie schon zuvor angedeutet, muß der Ergebnistyp einer Funktion keineswegs unbedingt `"void"` sein. In unserem Beispiel mit dem größten gemeinsamen Teiler (ggt) wäre es doch ausgesprochen praktisch, wenn die Funktion das Ergebnis nicht nur einfach auf den Bildschirm ausgeben, sondern es an das aufrufende Programm zurückgeben würde, damit man es im weiteren Programmablauf verwenden kann. Das geht im Prinzip ganz einfach:

```
int ggt(int i, int j)
{
  while (i)
  { int hilf = j % i;
    j = i;
    i = hilf;
  }
  return j;
}
```

Zunächst einmal wird als Ergebnistyp der Funktion `"int"` angegeben, es wird also eine Zahl zurückgegeben. Unser Algorithmus läuft dann genau wie bei der ersten Version, nur daß wir keine Ausgaben machen. Am Ende unserer Funktion steht dann eine `"return"`-Anweisung. Hier wird die Funktion beendet und das Ergebnis, also der Wert von `"j"`, an das aufrufende Programm zurückgegeben.

Dort können wir jetzt den Funktionsaufruf in Ausdrücken und Anweisungen benutzen, etwa so:

```
void main()
{ int a, b;
  cout « "Bitte zwei Zahlen eingeben: "; cin » a » b;
  int erg = ggt(a, b);
  cout » "Der ggt der Eingabe ist: " » erg
}
```

...oder auch so:

```
void main()
{ int a, b;
  cout « "Bitte zwei Zahlen eingeben: "; cin » a » b;
  cout » "Der ggt der Eingabe ist: " » ggt(a, b);
}
```

Da ein Funktionaufruf ein Ausdruck ist, kann er z. B. auch Argument einer anderen Funktion sein:

```
int Eingabe(int min, int max)
{ int input;

  for(;;)
  {
    cout « "Bitte eine Zahl zwischen " « min « " und " « max
    « "eingeben: ";
    cin » input;

    if (input >= min && input <= max)
      return input;

    cout « "Falsche Eingabe! \n";
  }
}

int ggt(int i, int j);       // wie oben

void main()
{
  cout « ggt(Eingabe(1,10000), Eingabe(1,10000));
}
```

In diesem Beispiel werden diverse Funktionsaufrufe ineinander verschachtelt. Zunächst wird im Hauptprogramm der Operator `"«"` ausgewertet, dessen Operanden (bzw. Argumente) das Objekt `"cout"` und der nachfolgende Ausdruck sind. Jener Ausdruck ist ein Aufruf der Funktion `"ggt"`, wobei als Argumente jeweils die Funktion `"Eingabe"` aufgerufen wird. Diese Funktion erwartet eine Eingabe des Benutzers und prüft dabei, ob sie im erwarteten Bereich liegt.

Es stellt für das Programm also kein Problem dar, die Auswertung eines Ausdrucks zu unterbrechen, eine Funktion auszuführen und dann den Ausdruck zu Ende zu berechnen.

Übrigens ist obiges Beispiel ziemlich gewagt programmiert, denn der C++-Sprachstandard sagt nichts darüber aus, in welcher Reihenfolge die Argumente einer Funktion ausgewertet werden, ob hier also zuerst der linke oder der rechte Aufruf von `"Eingabe"` erfolgt. Bei der Berechnung des größten gemeinsamen Teilers spielt die Reihenfolge der Parameter aber keine Rolle, so daß hier unterschiedliche Implementierungen kein wesentlich anderes Ergebnis liefern würden. MaxonC++ wertet die Argumente von links nach rechts aus, aber um zu anderen Implementierungen kompatibel zu bleiben, sollten Sie sich nie darauf verlassen.

Zum Abschluß noch einige Anmerkungen zur `"return"`-Anweisung:

- Eine Funktion kann beliebig viele `"return"`-Anweisungen haben. Die Ausführung einer Funktion wird durch die `"return"`-Anweisung sofort abgebrochen und der nachfolgende Wert zurückgegeben.
- Auch in einer Funktion mit `"void"`-Ergebnis kann `"return"` benutzt werden. In diesem Fall wird natürlich kein Ergebniswert angegeben, man schreibt also ganz einfach `"return;"`.

- Viele Programmierer setzen das Ergebnis einer `"return"`-Anweisung in runde Klammern, etwa `"return (2*j);"`, wodurch das Ganze wie ein Funktionsaufruf aussieht. Das darf man machen (es ist erlaubt, in und um Ausdrücke überflüssige Klammern zu setzen), aber es hat keinerlei Bedeutung. Auch in diesem Handbuch wird auf diese Klammern verzichtet, denn `"return"` ist nun einmal keine Funktion, sondern eine eigenständige Art von Anweisung.
- Der Ausdruck hinter `"return"` wird nach den üblichen Regeln für Initialisierungen in den Ergebnistyp der Funktion umgewandelt.

Es ist weit verbreitet (und leider auch erlaubt), bei der Deklaration einer Funktion überhaupt keinen Ergebnistyp anzugeben, z. B.

```
Anonym()
{ // irgendwas...
  return;
  // nochwas ...
  return 42;
}
```

Wenn eine solche Funktion in einem Ausdruck aufgerufen wird, wird davon ausgegangen, daß ihr Ergebnis `"int"` ist. Innerhalb einer Funktion ohne Rückgabetyp darf `"return"` wahlweise mit oder ohne Ausdruck aufgerufen werden, während sonst die Schreibweise ohne nachfolgenden Ausdruck nur in `"void"`-Funktionen und die mit Ausdruck ausschließlich in Funktionen mit einen anderen Ergebnistyp als `"void"` benutzt werden darf.

Deklarationen von Funktionen ohne Ergebnistyp sind schon wieder eine Erbschaft von C.

In diesem Zusammenhang sei noch einmal deutlich gesagt, daß C nicht, wie z. B. Pascal, zwischen `"Functions"` mit Ergebnis und `"Procedures"` ohne Ergebnis unterscheidet. Eine Funktion mit einem Ergebnis kann ganz normal als Anweisung aufgerufen werden, wobei ihr Resultat dann ignoriert wird:

```
int f()
{ cout « "f wurde aufgerufen.\n";
  return 26731;
}

void main()
{ int i = f();   // Ergebnis von "f" wird benutzt
  f();           // Ergebnis von "f" wird ignoriert
}
```

1.6.4 Funktionen und Gültigkeitsbereiche

1.6.4.1 Deklarationen auf Dateiebene

Jede Funktion hat ihren eigenen Gültigkeitsbereich. Das heißt, daß eine Funktion nicht auf Variablen oder sonstige Bezeichner einer anderen Funktion zugreifen kann - ein Bezeichner, der innerhalb einer Funktion definiert wird, ist außerhalb nicht sichtbar. Also kann man natürlich denselben

Namen in verschiedenen Funktionen definieren - diese Bezeichner haben dann für den Compiler absolut nichts miteinander zu tun.

Wenn Funktionen also gemeinsame Daten benötigen, muß die aufrufende Funktion der aufgerufenen die gewünschten Daten als Parameter übergeben. Dies ist oft nervig und manchmal gar nicht praktikabel, weshalb es noch eine weitere Möglichkeit gibt: Deklarationen auf Dateiebene. Ein Beispiel:

```
#include <stream.h>

int zahl1, zahl2, ergebnis;

void ggt()
{ int i = zahl1, j = zahl2;
  while (i)
  { int hilf = j % i;
    j = i;
    i = hilf;
  }
  ergebnis = j;
}

void main()
{ cout « "Bitte zwei Zahlen eingeben: ";
  cin » zahl1 » zahl2;
  ggt();
  cout « "ggT: " « ergebnis;
}
```

Die Variablen **"zahl1"**, **"zahl2"** und **"ergebnis"** werden auf Dateiebene (man sagt auch „global") definiert. Ihr Gültigkeitsbereich liegt damit noch über dem der Funktionen. Sie sind in allen Funktionen sichtbar, können dort aber natürlich auch überdefiniert werden.

Das kleine Programm simuliert Parameterübergaben im BASIC-Stil: Die Parameter werden in Variablen abgelegt, woraus das „Unterprogramm" sie übernimmt und das Ergebnis berechnet, das dann in einer globalen Variablen abgelegt wird. Damit ist das Programm wieder einmal ein Beispiel dafür, wie man nicht programmieren sollte. Es gibt aber tatsächlich Situationen, in denen man globale Variablen benutzen muß. Dann sollte man aber auf jeden Fall einen Kommentar an den Anfang der betreffenden Funktionen setzen, der auf solche „versteckte" Parameter hinweist.

Es ist nicht nötig, die globalen Variablen vor den Funktionen zu definieren. Funktions- und Variablendeklarationen dürfen in C beliebig gemischt werden. Da man aber jede Variable deklarieren muß, bevor man sie benutzen kann, ist es in der Regel am sinnvollsten, erst alle Variablen und dann die Funktionen zu deklarieren.

Übrigens werden auch die Objekte **"cin"** und **"cout"** in der Includedatei **"stream.h"** als globale Variablen deklariert, das erklärt dann auch, warum sie in allen Funktionen ohne weiteres zur Verfügung stehen.

1.6.4.2 Rekursive Funktionen

Eine Variable, die innerhalb einer Funktion definiert wird, wird bei jedem Aufuf der Funktion auf den „Stack" erzeugt und nach Beendigung der Funktion wieder gelöscht. Das hat dann gewichtige Konsequenzen, wenn eine Funktion sich selbst aufruft (man nennt sie dann „rekursive Funktion").

Ein fast schon klassisches Beispiel für Rekursion: Die Fakultät einer Zahl n wird oft deklariert als Produkt aller Zahlen von 1 bis n, z. B.

```
5! = 1*2*3*4*5 = 120
```

Es gibt aber eine noch viel schönere Definition: Man setzt `0! = 1` und für alle anderen Zahlen gilt:

```
n! = n * (n-1)!
```

Nach dieser Definition ist dann `5! = 5 * 4! = 5 * 4 * 3!` und so weiter. Und genau nach dieser Definition berechnet die folgende Funktion die Fakultät einer Zahl:

```
int fak(int n)
{ if (n == 0)
   return 1;
else
   return n * fak(n-1);
}
```

Es sollte Sie absolut nicht stören, daß die Funktion sich selbst aufruft, das tut sie nämlich nicht endlos oft (was man versehentlich auch öfters programmiert und dann zu einem Absturz führt, weil der Stack überläuft), sondern nur, bis die Variable `"n"` auf 0 heruntergezählt ist. Die Parametervariable `"n"` wird bei jedem einzelnen Aufruf neu erzeugt und anschließend wieder gelöscht, so daß die Rekursion nicht den Wert von `"n"` auf der aktuellen Ebene verändern kann. Genauso würde für weitere lokale Variablen, die man in `"fak"` deklarieren könnte, bei jeder Rekursion ein neuer Platz auf dem Stack eingerichtet. Damit gibt es keinen Unterschied zwischen einer Rekursion und einem gewöhnlichen Funktionsaufruf: Die lokalen Variablen einer Funktion werden in beiden Fällen garantiert nicht verändert, es sei denn, man wünscht das ausdrücklich - wie das geht, steht im Kapitel 2.

Prinzipiell kann man jede Schleife durch eine Rekursion ersetzen, so wie wir im letzten Beispiel eine simple Zählschleife zum Multiplizieren der ersten `"n"` Zahlen durch eine elegante rekursive Funktion ersetzt haben. Sie sollten es mit Rekursionen allerdings nicht übertreiben: Zum einen hat der Stack, auf dem bei jedem Funktionsaufruf Variablen und andere Daten abgelegt werden, nur eine begrenzte Größe, und wenn der überläuft, kommt der allseits beliebte Guru. Zum anderen benötigt ein Funktionsaufruf immer einiges an Zeit (Sie erinnern sich: Variablen müssen erzeugt und nachher wieder gelöscht werden, und auch sonst ist einiges internes Stack-Management nötig), so daß eine Schleifenstruktur oft weniger schön, aber stets schneller ist.

1.6.4.3 Deklarationen und Definitionen

Die Namen von Funktionen sind naheliegenderweise auf Dateiebene deklariert. Und spätestens jetzt ist es an der Zeit, einmal auf die Unterschiede zwischen einer „Deklaration" und einer „Definition" zu kommen.

Es ist nämlich durchaus möglich, eine Funktion zu deklarieren, ohne sie zu definieren. Betrachten wir einmal die beiden folgenden Funktionen:

```
int gerade (int);

int ungerade (int n)
{ if (n<0)
    return ungerade (-n);
  else
    return !gerade(n);
}

int gerade (int n)
{ if (n == 0)
    return 1
  else
    return ungerade(n-1);
}
```

Die beiden Funktionen stellen auf denkbar umständliche Art und Weise fest, ob eine Zahl gerade oder ungerade ist, nämlich rekursiv nach folgenden Regeln:

1. Eine negative Zahl ist ungerade, wenn die zugehörige positive Zahl ungerade ist.
2. Eine positive Zahl ist ungerade, wenn sie nicht gerade ist.
3. Null ist gerade.
4. Eine Zahl ist gerade, wenn die nächstkleinere Zahl ungerade ist.

Umständlicher geht es wirklich nicht, aber das steht hier nicht zur Debatte. Wesentlich ist vielmehr, daß die beiden Funktionen **"ungerade"** und **"gerade"** sich gegenseitig aufrufen. In C++ muß man aber jede Funktion deklarieren, bevor man sie benutzt (in C ist das anders, siehe auch den „Anachronismen"-Abschnitt). Deshalb wird zuerst die Funktion **"gerade"** nur deklariert: Ergebnistyp, Name und Parameterliste werden angegeben und das Ganze mit einem Semikolon beendet. Nun weiss der Compiler, was er mit dem Bezeichner **"gerade"** anzufangen hat: die Funktion **"gerade"** ist bekannt und darf deshalb aufgerufen werden. Man nennt eine solche Funktionsdeklaration ohne nachfolgende Definition auch „Prototyp" der Funktion.

Man beachte, daß hier in der Deklaration der Parameter den Typen **"int"**, aber keinen Namen hat. Man darf sie durchaus wie gewohnt benennen, aber diese Bezeichner haben dann keinerlei Bedeutung. Es ist sogar erlaubt, eine Funktion beliebig oft zu deklarieren und jedesmal andere Bezeichner für die Parameter zu wählen. Es kommt lediglich darauf an, daß die Datentypen und die Reihenfolge der Parameter gleich bleiben.

Nach der Deklaration und gleichzeitiger Definition der Funktion **"ungerade"** folgt im obigen Beispiel die Definition, also der Funktionsrumpf, der bisher nur deklarierten Funktion **"gerade"**. Hier muß wieder die komplette Parameterliste angegeben werden (bei **"forward"**-Deklarationen in Pascal ist das ja anders). Es ist zu empfehlen, den Parametern hier Namen zu geben, denn sonst

kann man sie in der Funktion ja nicht benutzen. Prinzipiell ist es aber erlaubt, namenlose Parameter zu deklarieren. Das kann z. B. sinnvoll sein, wenn man einen Parameter derzeit noch nicht benötigt, aber plant, die Funktion später so zu erweitern, daß ein zusätzlicher Parameter nötig wird. Dann kann man gleich in der Funktionsdeklaration durch einen namenlosen Parameter deutlich machen, daß eben dieser Parameter vorerst noch keine Bedeutung hat.

Selbstverständlich kann jede deklarierte Funktion nur einmal definiert werden, d. h. es kann nur einmal ein Anweisungsteil für jede Funktion geschreiben werden. Wenn eine Funktion deklariert, aber nicht definiert wird, nimmt der Compiler automatisch an, daß sie extern eingebunden werden soll. Deshalb meldet dann ggf. der Linker, nicht aber der Compiler einen Fehler.

Deklaration und Definition von Funktionen zu trennen, ist unumgänglich, wenn wie oben geschehen zwei Funktionen sich gegenseitig aufrufen. Aber auch sonst ist das oft sinnvoll: Viele Programmierer setzen an den Anfang eines Programms Deklarationen für sämtliche benutzte Funktionen. Dann darf man anschließend die Funktionedefinitionen in beliebiger Reihenfolge anordnen, muß also nicht zuerst die simplen Hilfsfunktionen und dann die gehaltvolleren Funktionen, die dann erstere benutzen, schreiben.

Funktionsdeklarationen ohne Definition benutzt man auch gerne in sog. „Header-Files" bei Projekten, die aus mehreren Modulen bestehen.

1.6.5 Überladen, Default-Parameter und mehr

1.6.5.1 Überladen von Funktionen

Ein recht nützliches Feature von C++ soll hier auch kurz erwähnt werden (das Thema ist in Wirklichkeit so umfangreich, daß ihm später ein eigenes Kapitel gewidnet werden soll): das „Überladen" von Funktionen.

Nehmen wir einmal an, Sie benötigen in einem Programm oft die Beträge von Zahlen, und zwar von Zahlen unterschiedlichster Datentypen. In gewöhnlichem C würden Sie dann folgendes schreiben:

```
int abs_i(int n)
{ if (n >= 0) return n;
  else return -n;
}

short abs_s(short n)
{ if (n >= 0) return n;
  else return -n;
}

float abs_f(float x)
{ if (x >= 0) return x;
  else return -x;
}

double abs_d(double x)
{ if (x >= 0) return x;
```

```
  else return -x;
}
```

Im nachfolgenden Programm müssen Sie dann immer solche Namen wie `"abs_f"` benutzen. Dabei wäre es doch viel schöner, wenn man immer nur `"abs"` sagen und der Compiler dann die richtige Funktion selbst heraussuchen würde. Und genau das ist in C++ möglich: es ist erlaubt, eine Funktion mit mehreren unterschiedlichen Parameterlisten zu deklarieren. Das könnte dann so aussehen:

```
int abs(int n)
{ if (n >= 0) return n;
  else return -n;
}

short abs(short n)
{ if (n >= 0) return n;
  else return -n;
}

// usw.
```

Wir haben es jetzt also mit mehreren Funktionen zu tun, die alle `"abs"` heißen. Beim Aufruf der Funktion `"abs"` betrachtet der Compiler dann den Typ des Arguments und ermittelt die `"abs"`-Funktion, zu der der Argumenttyp am besten paßt. Natürlich existieren präzise standardisierte Regeln, die festlegen, was in diesem Sinne „am besten passen" bedeutet. Diese Regeln möchte ich Ihnen an dieser Stelle aber noch nicht zumuten, zumal wir bis jetzt noch bei weitem nicht alle Datentypen kennen. Vorerst sollten Sie beim Aufruf einer überladenen Funktion darauf achten, daß die Typen der Argumente genau mit den Parametertypen der gewünschen Funktion übereinstimmen.

Ein Beispiel:

```
void f(int, double);

void f(double, int);

main()
{ f(1, 2);          // Fehler
  f(1.0, 2.0);   // Fehler
  f(1, 2.0);       // OK
}
```

Bei den beiden ersten Funktionsaufrufen wird der Compiler einen Fehler melden, denn er weiß nicht, welche Funktion hier jeweils gemeint ist. Der dritte Funktionsaufruf hingegen ist erlaubt, denn hier hat der erste Parameter den Typ `"int"` und der zweite den Typ `"double"`, so daß klar ist, welche der beiden Funktionen gemeint ist.

Es ist nicht möglich, den Ergebnistyp einer Funktion zu überladen, d. h. haben zwei Funktionsdeklarationen gleichen Namen UND gleiche Parameter, so müssen auch die Ergebnistypen gleich sein. Folgendes wäre also NICHT erlaubt:

```
int f();

 double f();  // Hier meldet der Compiler einen Fehler

main()
{ int i = f();
  double d = f();
}
```

Es ist zwar intuitiv klar, daß beim ersten Funktionsaufruf die `"int"`-Variante und beim zweiten die `"double"`-Version der Funktion `"f"` gemeint sind. C++ ermittelt die gewünschte Funktionsinstanz aber ausschließlich anhand der Argumente, ohne den „gewünschten" Ergebnistyp zu beachten. Der Grund ist einfach der, daß die Regeln für den Aufruf überladener Funktionen auch so schon kompliziert genug sind und sie niemand mehr so recht verstehen würde, wenn auch noch der Ergebnistyp in die Regel einginge.

Wie versprochen, verschone ich Sie vorerst noch mit den genauen Regeln für das Überladen von Funktionen. Wenn Sie beim Aufruf immer darauf achten, daß die Argumenttypen exakt mit den Parametertypen übereinstimmen, können Sie so viel gar nicht falsch machen. Zu guter Letzt sei noch darauf hingewiesen, daß man den Compiler in früheren C++-Versionen (1.0-Standard) eigens darauf hinweisen mußte, wenn man beabsichtigte, eine Funktion zu überladen. Das geschah dann mittels des Schlüsselworts `"overload"`, z. B. mußte man

```
overload Funktionsname;
```

deklarieren, bevor man die Funktion dieses Namens überladen durfte. Inzwischen ist C++ nicht mehr so kleinlich, und diese expliziten `"overload"`-Deklarationen sind auch gar nicht mehr Bestandteil des Sprachstandards. MaxonC++ akzeptiert solche Deklarationen immer noch, damit man ältere Quelltexte problemlos übersetzen kann, es ist aber nicht empfehlenswert, derartige Deklarationen heute noch zu verwenden.

1.6.5.2 Abkürzen durch Default-Argumente

Am Beispiel des Überladens von Funktionen haben Sie sicher schon erkannt, daß C++ eine wesentlich mächtigere Programmiersprache als C ist und daß sich der Umstieg auf jeden Fall lohnt. In diesem Abschnitt soll von noch einem praktischen Feature von C++ die Rede sein, nämlich Default-Parametern.

Manchmal hat eine Funktion so etwas wie einen „kanonischen" Parameter, dem fast immer dasselbe Argument übergeben wird. Dann nervt es ganz einfach, diesen Parameter bei jedem Aufruf anzugeben, nur weil an ein paar anderen Stellen ein anderer Wert dafür benötigt wird. Dann hat der C++-Programmierer gut Lachen und benutzt einfach einen Default-Parameter.

Man deklariert einen Default-Wert für ein Argument einer Funktion, indem man in der Funktionsdeklaration hinter den Parameter ein `"="` und den gewünschten Ausdruck (egal, ob nur eine Konstante oder eine komplexe Berechnung) setzt. Ein Beispiel:

```
void f(int i1, int i2 = 0);
```

Die so deklarierte Funktion kann dann wahlweise mit einem oder zwei Argumenten aufgerufen werden. Wenn nur ein Argument angegeben wird, z. B.

```
f(42);
```

setzt der Compiler für den zweiten Parameter automatisch `"0"` ein.

Es dürfen stets nur für den letzten bzw. die letzten Parameter einer Funktion Default-Argumente deklariert werden. Ungültig wäre deshalb

```
int f2(int a, short b=17, char c);  // Error
```

Aus technischen Gründen darf jedes Default-Argument auch nur einmal deklariert werden, auch wenn die Funktion mehrfach deklariert wird. Also wäre folgender Programmcode fehlerhaft:

```
void myfun(double = 1.0); // OK

void myfun(double = 1.0)  // Error: Redefinition of default argument
{ // mach' irgendwas;
}
```

Korrekt wäre hingegen überraschenderweise, bei der zweiten Deklaration das Default-Argument einfach wegzulassen. Der Compiler merkt sich dann das Argument der ersten Deklaration.

Welchen Sinn hat diese Regelung? Wäre es nicht übersichtlicher und leichter verständlich, Default-Argumente bei jeder Deklaration neu anzugeben?

Das wäre tatsächlich schöner, und einige frühe C++-Implementierungen erlauben dies auch (MaxonC++ aber nicht). Das Problem ist aber, daß man jedesmal dasselbe Argument angeben müßte, und es ist eine verblüffende Tatsache, daß es nicht so einfach ist, zu definieren, wann zwei Ausdrücke „gleich" sind. Nehmen wir einmal an, im obigen Beispiel hätten wir als Default Argument zunächst `"1.0/3.0"` gewählt. Bei der zweiten Deklaration geben wir nun `"0.333333"` an. Sind diese beiden Ausdrücke gleich? Auf einem Rechner mit bis zu 6-stelliger Rechengenauigkeit sind sie es, bei mehr als 6 Stellen Genauigkeit aber nicht. Noch komplizierter wird das Ganze dadurch, daß man als Default Parameter nicht nur Konstanten, sondern nahezu beliebige Ausdrücke deklarieren kann. Sind z. B. die Ausdrücke `"a+(b+1)"` und `"(a+b)+1"` identisch? Jedenfalls würde eine verbindliche Definition, wann zwei Ausdrücke identisch sind, ziemlich schwierig. Also hat man es dabei belassen, jedes Default Argument nur einmal angeben zu können.

Dafür darf man aber auch nachträglich zusätzliche Default-Argumente deklarieren. Ein Beispiel:

```
void seltsam(int, char, double);
```

```
void seltsam(int, char, double = 1.0);

void seltsam(int, char = '?', double);

void seltsam(int = 42, char, double);
```

Nach der ersten Deklaration benötigt `"seltsam"` noch drei Argumente. Nun wird nachträglich ein Default für das letzte Argument deklariert, so daß wir die Funktion nun auch mit nur zwei Argumenten aufrufen können. Bei der dritten Deklaration bekommt auch noch der `"char"`-Parameter einen Defaultwert. Man beachte, daß hier scheinbar ein Default-Argument für einen Parameter, der nicht am Ende steht, deklariert wird, und das ist bekanntlich verboten. In Wirklichkeit weiß der Compiler natürlich, daß der letzte Parameter schon ein Default-Argument besitzt, und so hat alles seine Richtigkeit.

Zu guter Letzt wird auch noch ein Default-Argument für den ersten Parameter hinzugefügt. Jetzt haben wir die freie Auswahl, ob wir `"seltsam"` nun kein, ein, zwei oder drei Argumente übergeben.

Eine kleine Einschränkung sei hier noch erwähnt: Der Ausdruck, den man als Default-Argument setzt, darf nicht Werte der anderen Argumente benutzen, und zwar (unter anderem) deshalb, weil nicht definiert ist, in welcher Reihenfolge Argumente ausgewertet werden. Wenn nun ein Default-Argument von einem anderen Argument abhängt, das noch nicht berechnet wurde, wäre das Ergebnis ja undefiniert.

Also klopft Ihnen der Compiler auf die Finger, wenn Sie so etwas versuchen:

```
int multi(int a, int b=a) // Falsch!
{ return a*b;
}
```

Die Intuition dieser Funktion ist einsichtig: Wenn sie zwei Argumente hat, berechnet sie deren Produkt, bei nur einem Argument dessen Quadrat. Falls man auf derartige „Default-Argumente" absolut nicht verzichten will, kann man sie durch Überladen der Funktion „simulieren":

```
int multi(int a, int b)
{ return a*b;
}

int multi(int a)
{ return multi(a,a);
}
```

Das kostet zwar etwas mehr Laufzeit (bei nur einem Argument wird das Ergebnis nicht direkt berechnet, sondern noch eine zweite Funktion aufgerufen), aber es funktioniert.

1.6.5.3 Variable Parameterlisten

C++ hat von C auch durchaus ein paar nützliche Erbschaften übernommen, die zwar nicht schön, aber trotzdem hin und wieder praktisch sind. Dazu gehören die variablen Parameterlisten, mit

denen man einer Funktion beliebig viele Parameter beliebigen Typs übergeben kann. Dazu deklariert man als letzten Parameter einer Funktion die Ellipse „...":

```
void int_und_dann_egal(int i, ...);
```

Diese Funktion erwartet als erstes Argument ein `"int"`, und danach kann man beim Aufruf Beliebiges setzen, z. B.

```
int_und_dann_egal(17, 4.0, 'x', 93L, 0x686b);
```

oder auch ganz kurz:

```
int_und_dann_egal(2);
```

Natürlich haben die zusätzlichen Parameter dann für die Funktion keinen Namen, so daß die Funktion darauf nicht direkt zugreifen kann, sondern nur mit Hilfe der Funktionen, die in `"<stdarg.h>"` deklariert werden. Näheres über dieses (und alle anderen) Includefile erfahren Sie im Referenzteil dieses Handbuchs.

1.6.6 Anachronismen aus der Mottenkiste

1.6.6.1 Umschalten in den C-Modus

Im uralten „Kernigham & Richie"-Standard für die Programmiersprache C waren Parameterlisten von Funktionen ausgesprochen seltsam definiert, und leider wirkt sich das noch bis in ANSI C fort. Bei der Einführung von C++ hat man diese alten Zöpfe lobenswerterweise abgeschnitten, aber das ist dann auch die Hauptursache für Inkompatiblitäten zwischen ANSI C und C++.

Diese „Erbschaften" von Uralt-C sind so verstaubt und sinnlos, daß sie von MaxonC++ (in Übereinstimmung mit dem C++-Standard) nicht mehr unterstützt werden, jedenfalls nicht im C++-Modus. Allerdings sind sie immer noch Bestandteil des ANSI-C-Standards, übrigens mit gutem Grund, denn es haut mich immer wieder um, wenn ich sehe, wie viele C-Programmierer diese antiquierten Features noch benutzen.

Jedenfalls möchte ich Sie noch einmal darauf hinweisen, daß es wirklich keinen vernünftigen Grund gibt, das im folgenden Beschriebene zu benutzen. Ich schreibe dieses Kapitel überhaupt aus nur drei Gründen:

1. MaxonC++ kann es, also soll es auch dokumentiert werden.
2. Man braucht diese alten Features, wenn man einen älteren Quelltext mit MaxonC++ übersetzen will.
3. Vielleicht versuchen Sie einmal, ein fremdes Listing zu verstehen, und dann sollen Sie sich nicht über eine gewisse seltsame Syntax wundern.

Bevor ich Ihnen verrate, worum es eigentlich geht, erzähle ich Ihnen noch kurz, wie Sie unter MaxonC++ in den ANSI-Modus gelangen. Bestimmt haben Sie schon das Pull-Down-Menü **„Optionen/Compiler/C++"** entdeckt. Wenn man diesen Menüpunkt ausschaltet, wird das Programm nach

den ANSI-C-Sprachregeln übersetzt. Aber man kann auch im Programmtext den Sprachstandard wählen und sogar an beliebiger Stelle zwischen C und C++ umschalten, und zwar mit einem `"pragma"`.

`"#pragma"`-Zeilen werden, genau wie `"#include"`, vom Preprozessor interpretiert und ausgeführt. Mit einem `"#pragma"` realisiert man dabei compiler-spezifische Features, die ausdrücklich nicht standardisiert sind. In MaxonC++ schaltet die Zeile

```
#pragma -
```

den Compiler auf ANSI-C-Emulation um, und

```
#pragma +
```

ermöglicht danach wieder die Programmierung in C++.

Jeder C-Compiler, der zu ANSI C kompatibel ist, kennt übrigens `"pragma"`-Zeilen, nur das, was dahinter steht und was es bedeuten soll, ist nicht im C-Standard festgelegt, denn damit soll man ja eben implementationsabhängige Funktionen ansprechen. Deshalb ist festgelegt, daß der Compiler keinen Fehler melden darf, wenn er auf ein unbekanntes `"pragma"` trifft. Also sollte jeder andere ANSI-C-Compiler ein `"#pragma -"` aus einem unter MaxonC++ entwickelten Programm ignorieren, während MaxonC++ Dinge wie `"#pragma Hallo!"` ganz einfach nicht beachtet.

1.6.6.2 Parameterlisten im alten Stil

Und nun lasse ich die Katze aus dem Sack: Bei den oben erwähnten Uralt-Features handelt es sich um die alte Syntax für Parameterlisten und die Konsequenzen, die das für ANSI C bis heute hat.

Ursprünglich wurden die Datentypen von Argumenten überhaupt nicht beachtet. Rief man eine Funktion auf, etwa

```
f(17, 4.0);
```

so legte der Compiler die `"int"`-Zahl `17` und die `"double"`-Konstante `"4.0"` auf den Stack und rief dann die Funktion `"f"` auf, ganz egal, was für Argumente `"f"` in Wirklichkeit erwartet. Das können Sie heute auch noch in C++ simulieren, indem Sie `"f"` mit einer vollkommen variablen Argumentenliste deklarieren:

```
int f(...);
```

In jenen alten Tagen mußte man eine Funktion dann auch überhaupt nicht deklarieren, bevor man sie aufrief - die Argumenttypen waren ja sowieso egal, und wenn nichts anderes deklariert wurde, wurde als Ergebnistyp `"int"` angenommen. Wenn man eine Funktion aber trotzdem deklarierte, sah das ganz einfach so aus:

```
int f();
```

Man durfte sogar den Ergebnistyp weglassen, also

```
f();
```

deklarieren, wobei dann als Ergebnis im Zweifelsfall `"int"` angenommen wurde. Eine solche Deklaration hat natürlich niemand geschrieben, denn da wird ja nichts anderes gesagt, als der Compiler ohnehin schon vermutet, nämlich daß die Funktion `"f"` existiert, irgendwelche nicht näher bekannten Argumente hat und entweder `"int"` oder gar nichts zurückgibt.

Bei der Definition der Funktion war es dann aber nötig, den Parametern Namen zu geben. Das sah dann so aus, daß nur die Namen in die Parameterliste gesetzt wurden. Zwischen der Funktionsdeklaration und dem Funktionsrumpf konnte man dann die Datentypen der Parameter deklarieren - wenn für einen kein Typ angegeben wurde, so wurde er als `"int"` betrachtet.

Ein Beispiel:

```
#pragma -    /* Das folgende akzeptiert MaxonC++ nur im ANSI-Modus */

int uralt(i,j,k,d,c)
   int i, j;
   double d;
   char c;
{
   d = i+j+k;
   return (int)d + c;
}
```

Aber glauben Sie nicht, der Compiler hätte wenigstens nach dieser Funktionsdefinition die Argumenttypen beachtet, oh nein! Auch danach blieben die Typen der Parameter Privatangelegenheit der Funktion, und beim Funktionsaufruf konnte der Programmierer nur beten, daß er immer die richtigen Datentypen erwischte, sonst bekam die Funktion unsinnige Parameter. Ach ja, das waren noch Zeiten, als Programmierer noch echte Männer waren und souverän die Fallgruben, die C ihnen grub, umgingen...

Übrigens kamen in letzter Zeit Gerüchte auf, daß das alte C eigentlich nur ein Scherz sein sollte. Demnach wollten Kernighan und Richie eine möglichst konfuse Parodie auf Pascal schreiben und das Ganze dann den Russen unterjubeln, um deren technologischen Rückstand weiter zu vergrößern. Ebenso soll UNIX ursprünglich nur eine Parodie auf MULTICS gewesen sein, aber dann kamen einige Verrückte und begannen, unter UNIX in C tatsächlich zu programmieren... Aber lassen wir das.

1.6.6.3 Von der Steinzeit in die ANSI-Ära

Mit der Einführung von ANSI C wurde dann einiges besser. Endlich wurden die Parameterlisten, die man schon von C++ kannte, in C aufgenommen. Sie haben durchaus richtig gelesen, denn C++ gab es schon vor ANSI C, wenn auch in einer etwas abgespeckten Version. Dabei gab es dann aber ein kleines Problemchen: das Ganze sollte auf jeden Fall kompatibel bleiben. Also blieben die alten Parameterlisten 'drin, und der Programmierer hat in ANSI C bis heute die Wahl, ob er nun im neuen oder alten Stil schreiben will. Es wird zwar allgemein empfohlen, die neue Form zu benutzen (was ja auch das einzig Sinnvolle ist), aber das interessiert erfahrungsgemäß den echten Hard-Core-C-Coder nicht.

Um die Kompatiblität zum alten C-Standard zu erhalten, mußte man definieren, daß die scheinbar leere Parameterliste `"()"` weiterhin für „beliebige Parameter ohne Typprüfung" stehen sollte - also ein deutlicher Unterschied zu C++. Eine Funktion, die wirklich keine Parameter haben soll, deklariert man in ANSI C, indem man in die Parameterliste `"void"` schreibt, z. B.

```
int fun(void);
```

Das sieht komisch aus und nervt, deshalb ist es allgemein verbreitet, für parameterlose Funktionen einfach `"()"` zu schreiben und dann selbst darauf zu achten, daß man keine überflüssigen Argumente übergibt.

MaxonC++ akzeptiert die `"(void)"`-Schreibweise auch im C++-Modus, damit man leichter gemeinsame Header-Files für C und C++ schreiben kann. Dem C++-Standard entspricht dies genaugenommen nicht, aber es ist dem Implementator freigestellt, derartige Anachronismen einzubauen. Wenn man unter C++ Parameterlisten im alten Stil benutzt, meckert MaxonC++ und bittet darum, in den C-Modus umzuschalten.

Zusammenfassend möchte ich zusammenfassenderweise zusammenfassen, daß die leere Parameterliste `"()"` je nach Modus entweder `"(...)"` oder `"(void)"` entspricht.

1.6.7 Einige Standardfunktionen

1.6.7.1 Allgemeines

Der C-Programmierer soll das Rad nicht jedesmal neu erfinden. Deshalb gehört zu jedem C-Entwicklungssystem eine umfangreiche Sammlung von Bibliotheksfunktionen für alle Lebenslagen. Diese werden im Referenzteil dieses Handbuchs ausführlich und systematisch beschrieben. Deshalb soll dieser Abschnitt Ihnen nur einen kleinen Überblick geben, damit Sie bei Bedarf besser Bescheid wissen, wo Sie zu suchen haben.

C hat im Gegensatz zu Pascal, Basic und anderen Programmiersprachen keine „eingebauten" Funktionen (in Pascal sind Funktionen wie `"writeln"` oder `"abs"` von Anfang an vorgegeben). In C ist das so konzipiert, daß man immer Includefiles (so wie `"stream.h"`), in denen dann die Funktionsdeklarationen stehen, benutzen muß. Der Linker bindet dann die benutzten Funktionen ein.

Erfreulicherweise gehören zur Sprachdefinition von ANSI C auch genormte Bibliotheken, so daß C-Programme theoretisch problemlos auf andere Rechner portiert werden können. Bei C++ ist man leider noch nicht so weit: hier gibt es nur als Konvention die Datei `"stream.h"`, deren Inhalt aber so genau gar nicht festgelegt ist.

Außerdem gibt es noch ein paar Spezialfunktionen von MaxonC++ und last not least die Amiga-Betriebssystemfunktionen. In diesem Abschnitt soll aber vor allem von den Standardfunktionen die Rede sein.

1.6.7.2 Ein- und Ausgabe: <stdio.h>

Die Includedatei `"stdio.h"` deklariert die Funktionen, die unter C zur Ein- und Ausgabe von Daten dienen (da man in C keine Operatoren überladen kann, fallen `"cout"` und `"cin"` hier flach).

Die wohl bekannteste C-Funktion ist `"printf"`, wobei Sie das `"print"` wahrscheinlich schon aus BASIC-Tagen kennen und das `"f"` für „Format" steht. Also dient `"printf"` zur universellen formatierten Ausgabe von Daten.

Als ersten Parameter erwartet `"printf"` einen sog. Formatstring. Das ist eine beliebige Zeichenkette, die dann im Prinzip auch ausgegeben wird, nur daß das Zeichen `'%'` dabei eine besondere Bedeutung hat: `'%'`, gefolgt von Formatangaben und einem Buchstaben, fügt an dieser Stelle des Strings Daten ein, die man hinter dem Formatstring angibt.

Ein Beispiel:

```
#include <stdio.h>

void main()
{ int a = 17, b = 4;
  printf("Die Summe von %d und %d ist: %d\n", a, b, a+b);
}
```

Wo immer `"%d"` in der Zeichenkette auftritt, nimmt `"printf"` das nächste Element aus der Parameterliste und gibt es an dieser Stelle als `"int"` aus. Dabei gibt es noch etliche Variationen: Bei `"%x"` erfolgt die Ausgabe in Hexadezimal, `"%hd"` bzw. `"%hx"` geben entsprechend `"short"`- oder `"unsigned short"`-Daten aus, `"%e"`, `"%f"` und `"%g"` geben `"double"`-Zahlen in unterschiedlichen Formaten aus usw. Dabei programmieren Sie übrigens ohne Netz und doppelten Boden, denn die Typen der Parameter hinter dem Formatstring werden nicht geprüft. Wenn Sie also einen „short int"-Wert mit `"%d"` statt `"%hd"` ausgeben, werden Sie unsinnige Ausgaben erhalten.

Alles in allem ist `"printf"` eine sehr mächtige Funktion, die weitaus flexiblere Ausgaben als `"cout"` ermöglicht, manchmal aber auch etwas komplizierter zu handhaben ist.

Es gibt dann noch die Funktion `"scanf"` für formatierte Eingaben. Ihr muß man Zeiger auf die einzulesenden Variablen übergeben, aber Zeiger werden erst im nächsten Kapitel behandelt.

Vor allem enthält `"stdio.h"` aber die gesamte Dateibehandlung. Die Funktion `"fopen"` öffnet eine Datei, `"fclose"` schließt sie, und mit `"fprintf"` und `"fscanf"` kann man formatiert in eine Datei schreiben bzw. aus ihr lesen.

1.6.7.3 Zeichen, Zeichenketten und Bytefolgen: <ctype.h> und <string.h>

Die Datei `"ctype.h"` deklariert kleine Funktionen auf Zeichen. Beispielsweise liefert die Funktion `"isalpha(c)"` den logischen Wert „wahr", wenn ihr Argument ein Buchstabe ist. Entsprechend stellen `"islower(c)"` und `"isupper(c)"` fest, ob ihr Argument ein Klein- bzw. Großbuchstabe ist. Entsprechende Funktionen gibt es auch für den Test auf Ziffer, Steuerzeichen und anderes.

Diese Funktionen sind vor allem deshalb wichtig, weil auch heute nicht jede (aber fast jede) Maschine den ASCII-Code benutzt. In ASCII könnte man

```
if (isupper(c))
```

ersetzen durch

```
if (c >= 'A' && c <= 'Z')
```

Das muß aber auf Nicht-ASCII-Systemen keineswegs so sein. Auf diese Weise verbessern die `<ctype.h>`-Funktionen die Portabilität von Programmen.

Nützlich sind auch `"tolower(c)"` und `"toupper(c)"`: Erstere wandelt ihr Argument, wenn es ein Großbuchgstabe ist, in einen Kleinbuchstaben und gibt es andernfalls unverändert zurück, letztere wandelt entsprechend Klein- in Großbuchstaben.

In `<string.h>` stehen die elementaren Funktionen für die Stringbehandlung von ANSI C, z. B. Kopieren, Aneinanderhängen oder Vergleichen von Zeichenketten. Zur näheren Erläuterung dieser Funktionen müßten Sie sich aber mit Zeigern und Vektoren auskennen, aber das steht erst im nächsten Kapitel.

Aus selbigem Grund sollen die `"mem"`-Funktionen, die ebenfalls in `"string.h"` deklariert werden, hier nur kurz erwähnt werden. Sie stellen Operationen auf Speicherblöcken dar.

1.6.7.4 Fließkommafunktionen: <math.h>

Wenn Sie die Includedatei `"<math.h>"` in Ihr Programm aufnehmen, stehen Ihnen die wichtigsten mathematischen Funktionen zur Verfügung. Vor allem werden dort `"double"`-Funktionen wie

```
double sin (double)
```

deklariert, die - Sie werden es erraten haben - den Sinus ihres Arguments berechnet. Entsprechend berechnen die Funktionen `"cos"` den Cosinus und `"tan"` den Tangens. Bei diesen trigonometrischen Funktionen sind Winkel stets im Bogenmaß (Radian) anzugeben, d. h. 360 Winkelgrad entsprechen 2*Pi.

Natürlich gibt es hier neben vielen anderen auch die Funktionen `"exp"` (für die Exponentialfunktion), `"log"` (natürlicher Logarithmus), `"sqrt"` (Quadratwurzel) und `"fabs"` (Betrag). `"floor"` rundet ab (liefert also die nächstkleinere ganze Zahl), wobei das Ergebnis aber trotzdem ein `"double"`-Wert ist, und `"ceil"` rundet entsprechend auf.

MaxonC^{++} bietet in `"<math.h>"` außerdem noch einige Funktionen, die Zahlen in ihre Zeichendarstellung verwandeln.

1.6.7.5 Die C-Fundgrube: <stdlib.h>

Alle wichtigen Funktionen, die sich unter keine der bisherigen Rubriken einordnen ließen, wurden in `<stdlib.h>` untergebracht. Besonders wichtig ist hier wohl die Funktion

```
void exit(int)
```

Sie bricht die Programmausführung ab, als wäre das Ende der Funktion `"main"` erreicht worden, und gibt eine Zahl (z. B. einen Fehlercode) an das aufrufende System zurück.

Häufig braucht man auch die Funktionen `"abs"` und `"labs"`, die den Betrag eines `"int"`- bzw. `"long int"`-Parameters berechnen. Wie Sie sehen, beherrscht ANSI C kein Überladen von Funktionen...

Praktisch ist auch die Funktion `"rand()"`, die eine ganzzahlige Zufallszahl im Bereich von `0` bis `32767` berechnet. In Wirklichkeit handelt es sich dabei nicht um Zufallszahlen, sondern um eine Zahlenfolge, die nur so aussieht, als wäre sie zufällig. Deshalb ist es auch nötig, den Startwert dieser Folge vor der ersten Benutzung von `"rand"` mit der Funktion `"srand(unsigned int)"` zu setzen, meist benutzt man hier die Systemzeit oder etwas Ähnliches.

MaxonC++ bietet Ihnen hier noch zwei andere Funktionen, oder sagen wir besser: eine überladene Funktion:

```
double Random()
```

liefert eine Fließkomma-Zufallszahl im Bereich von `0` bis `1`, und

```
unsigned int Random(unsigned int n)
```

berechnet eine Zufallszahl im Bereich von `0` bis `n-1`. So kann man z. B. mit `"Random(6)+1"` einen Würfel simulieren. Die beiden „Random"-Funktionen benutzen nicht nur ein Standardverfahren, sondern als zusätzliches Zufallselement die aktuelle Rasterposition des Elektronenstrahls. Deshalb müssen sie auch nicht mit `"srand"` initialisiert werden.

Außerdem enthält `"stdlib.h"` Funktionen, die Zeichenketten in Zahlen verwandelt, und die ANSI-C-Funktionen zur Speicherverwaltung (z. B. `"malloc"`). Letztere dürfen Sie aber getrost vergessen, denn in C++ geht das viel eleganter. Nützlich ist auch die Funktion `"qsort"`, die ein Datenfeld nach dem Quicksort-Verfahren sortiert.

Dieser Abschnitt hat Ihnen hoffentlich einen kleinen Überblick über die Funktionen von ANSI C gegeben. Alles weitere finden Sie im Referenzteil dieses Handbuchs.

2. Datentypen und Deklarationen

2.1 Vektoren, Zeiger und Referenzen

2.1.1 Pointer

2.1.1.1 Von Objekten und Adressen

Ein zentrales Konzept der Programmierung in C (und Quelle ewiger Verzweiflung für Anfänger) sind die Zeiger, auf neudeutsch auch „Pointer" genannt. Um in diesem Zusammenhang Konfusion von Anfang an zu vermeiden, sollten wir uns erst einmal genauer ansehen, was im Speicher so los ist.

Eine Variable in C ist ein Datenobjekt, das irgendwo im Speicher liegt. Wahrscheinlich (hoffentlich) wissen Sie, wieviel Bytes RAM Ihr Amiga hat. Damit der Prozessor damit auch etwas anfangen kann, sind alle diese Bytes durchnumeriert - das nennt man die „Adresse" eines Speicherbytes. Beim 68000-er Prozessor, der (eventuell in verbesserter Ausführung) in jedem Amiga steckt, ist eine Adresse 32 Bit breit, genau wie z. B. ein `"int"` in MaxonC++. Theoretisch kann man damit also gediegene 4 Gigabyte Speicher adressieren (Adressen betrachtet man übrigens normalerweise als vorzeichenlos). Jeder real existierende Amiga hat natürlich weniger RAM, so daß viele Adressen nicht benutzt werden. Trotzdem ist eine Adresse stets 32 Bit groß.

Nun ist ein Byte nicht gerade viel, nämlich gerade genug, um darin ein Zeichen (z. B. eine `"char"`-Variable) abzulegen. „Breitere" Datentypen nehmen dann entsprechend mehrere Bytes ein, die stets an aufeinanderfolgenden Adressen liegen. Als Adresse dieses größeren Datengebildes gilt dabei stets die Adresse des ersten Bytes. Wenn also ein `"int"`-Objekt die vier Speicherbytes von `0x00053ac4` bis einschließlich `0x00053ac7` belegt (Adressen gibt man gerne als hexadezimale Zahlen an), sagt man einfach, es liegt an Adresse `0x00053ac4`.

Nun könnte man ganz einfach eine Speicheradresse eines Objekts in einer Variablen des Typs `"unsigned int"` abspeichern und dies dann als Verweis oder Zeiger auf die Speicherstelle benutzen. Weil das Zeigerkonzept aber so enorm wichtig und nützlich ist, wird es von C noch viel weitreichender unterstützt: Man kann sich zu jedem Datentypen `"X"` einen Pointertypen `"Zeiger auf X"` deklarieren. Der unäre Operator `"&"`, der einem Objekt vorangestellt wird, liefert die Adresse dieses Objekts, die dann automatisch vom entsprechenden Pointertyp ist. Mit dem unären `"*"`-Operator erhält man dann umgekehrt wieder aus einem Zeigerausdruck das dadurch „referierte" Datenobjekt.

Klingt das kompliziert? Nun, dann wollen wir gleich zur Praxis übergehen.

2.1.1.2 Zeiger: Deklaration und elementare Operationen

Wie oben bereits angedeutet, gibt es nicht nur einen einzigen Datentyp „Zeiger", sondern zu jedem einzelnen Datentyp einen entsprechenden Pointertyp. Folglich gibt es auch „Zeiger auf Zeiger" und „Zeiger auf Zeiger auf Zeiger auf...", also unendlich viele Datentypen. Natürlich kennt der Compiler

nicht unendlich viele vordefinierte Namen für alle diese Typen. Deshalb müssen wir hier erstmals einen Datentypen selbst definieren, indem wir ihn beschreiben.

Ein (übrigens nicht sehr schlaues) Prinzip der C-Syntax ist, daß die Deklaration einer Variablen möglichst genau wie ihre Benutzung aussehen soll. Nehmen wir an, `"p1"` soll ein Zeiger auf einen `"double"`-Wert sein. Dann ist, wie oben bereits angedeutet, `"*p1"` eben das `"double"`-Wort, auf das `"p1"` gerade zeigt. Deshalb deklarieren wir die Zeigervariable `"p1"` folgendermaßen:

```
double *p1;
```

Das ist relativ einleuchtend: wir deklarieren nämlich, daß `"*p1"` vom Typ `"double"` sein soll, folglich ist `"p1"` ein Zeiger auf `"double"`. Man sagt auch: `p1` ist vom Datentyp `"double*"`. Deshalb sollten Sie aber nicht in Versuchung geraten, mehrere Pointervariablen so zu deklarieren:

```
int* i1, i2;  // Vorsicht!
```

Hier gehört das `"*"` ausschließlich zu `"i1"`. Demnach deklariert diese Zeile `"i1"` als Zeiger auf `"int"` und `"i2"` als `"int"`. Mehrere Zeiger muß man so deklarieren:

```
int *i1, *i2, *i3;
```

Unmittelbar nach der Deklaration ist ein Zeiger, genau wie alle anderen Variablen auch, undefiniert und zeigt „irgendwohin". Bevor man ihn benutzt, sollte man ihm also unbedingt einen sinnvollen Wert geben, z. B. auf eine andere Variable zeigen lassen. Das könnte so aussehen:

```
#include <stream.h>

void main()
{  int i1 = 41, i2 = 98; // Zwei ganz normale int-Variablen

   int *ip = &i1;        // Jetzt zeigt ip auf i1

   cout « *ip;           // Gibt den Wert von i1 aus

   i1++;                 i1 wird verändert

   cout « *ip;           // Jetzt wird der NEUE Wert ausgegeben

   ip = &i2;             // Nun zeigt ip auf i2

   *ip = 26731;          // Wert von i2 wird über ip geändert

   cout « i2;            // Also Ausgabe "26731"
}
```

Der Operator `"&"` liefert die Adresse eines Objekts, wobei das Ergebnis den jeweiligen Pointertyp hat. Also ist im obigen Beispiel `"&i1"` ein Ausdruck des Typs `"int*"` und somit eine passende Initialisierung für die Variable `"ip"`. Nun zeigt `"ip"` also bis auf weiteres auf `"i1"`, und wir können `"*ip"` synonym zu `"i1"` verwenden. Wenn wir also `"*ip"` mit `"cout"` ausgeben, wird immer

der Wert ausgegeben, denn `"i1"` gerade hat, nicht etwa der, den `"i1"` im Zeitpunkt der Zuweisung hatte.

Nicht nur der Wert des Objekts, auf das ein Pointer zeigt, sondern auch der Wert eines Pointers selbst kann verändert werden. Um das zu demonstrieren, wird im Beispielprogramm `"ip"` anschließend die Adresse von `"i2"` zugewiesen. Und zu allem Überfluß wird danach auch noch der Wert von `"*ip"` verändert! Da `"ip"` jetzt aber auf `"i2"` verweist, ist klar, was das bedeutet: nun ändert sich der Wert von `"i2"`, ohne daß wir diesen Variablennamen auch nur erwähnen!

Daß Variablen auf diese Weise unter einem anderen Namen benutzt werden können und man außerdem noch zwischen einer Zeigervariablen und dem von ihr referierten Objekt unterscheidet, ist für Anfänger oft so verwirrend, daß sie beschließen, auf Pointer ganz zu verzichten und so viele Dinge niemals realisieren können. Ich empfehle Ihnen dringend, sich nicht abschrecken zu lassen, wenn man sich einmal etwas daran gewöhnt hat, ist das alles absolut trivial.

2.1.1.3 Zeiger als Funktionsparameter

In C benutzt man Zeiger gern, um Variablenwerte von Funktionen verändern zu lassen, denn C kennt (im Gegensatz zu C++) keine „VAR-Parameter" wie Pascal. Ein kleines Beispiel:

```
#include <stream.h>

void Quadriere (int *par)
{
   *par = *par**par;  // Schlechter Stil, übersichtlicher wäre:
                      // *par = (*par) * (*par)
}

main()
{  int i = 17;

   Quadriere (&i);        // Wir übergeben nicht "i", sondern einen
                          // Zeiger darauf!
   cout « "17^2 ist: " « i;
}
```

Die Funktion `"Quadriere"` ist als `"void"` deklariert, liefert also keinen Wert direkt zurück. Vielmehr erwartet sie als Argument einen Zeiger auf ein `"int"`-Objekt, dessen Wert sie quadriert und dann in eben jenem Objekt ablegt. Die chaotische Ansammlung von `"*"`-chen (teils ist der unäre Inhaltsoperator, teils der binäre Multiplikationsoperator gemeint) soll hier nur illustrieren, daß man ein Programm mit Pointeroperationen leicht unübersichtlich gestalten kann. In der Praxis sollte man wohl besser Klammern zur Gliederung verwenden.

Beim Aufruf der Funktion muß man dann natürlich nicht die Variable selbst, sondern einen Zeiger darauf als Argument übergeben. Diese Art der Parameterübergabe ist also erheblich umständlicher als ein VAR-Parameter in Pascal. Aber keine Panik: C++ bietet, als Erweiterung gegenüber C, Referenztypen, mit denen man genau das viel schöner realisieren kann.

Übrigens kann eine Funktion auch einen Zeiger zurückgeben:

```
int *pfun()
{ int j = 99;
  return &j;          // Vorsicht!!!
}

void main()
{ cout « *pfun();  // Ergebnis undefiniert!
}
```

Die Funktion `"pfun"` liefert hier als Ergebnis einen Zeiger auf eine lokale Variable, aber nach Beendigung der Funktion existiert diese Variable nicht mehr, so daß das Funktionsergebnis hier lediglich dahin zeigt, wo die Variable `"j"` einmal gewesen ist. Es gibt keine Garantie, daß dieser Speicherplatz nach dem Funktionsaufruf nicht sofort wieder überschrieben wird. Also sollte man sich immer gründlich überlegen, welchen Zeiger man aus Funktionen gefahrlos hinausreichen kann.

2.1.1.4 Zeigertypen

Bisher habe ich Ihnen diskret verschwiegen, daß ein `"int*"` etwas ganz anderes als z. B. ein `"char*"` ist. Zwar werden beide intern als 32 Bit breite Adresse dargestellt, aber trotzdem ist es so ohne weiteres nicht erlaubt, unterschiedliche Zeiger einander zuzuweisen:

```
int *i, *ii;
char *c;

ii = i;           // Erlaubt, denn gleiche Typen
c = i;              // Verboten!
c = (char*) i;// So geht's.
```

Wenn man einen Zeiger in einen anderen Zeigertyp umwandeln will, muß man das also (z. B. mit einem Cast) deutlich sagen, denn auf diese Weise kann man erheblichen Schaden anrichten - den Rechner zum Absturz zu bringen, ist nicht weiter schwierig! Immerhin ist ein `"int*"` ein Zeiger auf das erste von vier Bytes, während ein `"char*"` auf ein einzelnes Byte zeigt. Würde man im letzten Beispiel eine Zuweisung an `"*c"` vornehmen, so würden also das erste Byte von `"*i"` verändert und die drei anderen unverändert bleiben - also wäre der Wert von `"*i"` irgendetwas (wahrscheinlich) Sinnloses. Noch fataler wäre folgende Zuweisungsfolge:

```
// c zeigt auf irgend so ein char
i = (int*) c;
*i = 4711;     // PANIK!
```

Die Zuweisung an `"*i"` verändert nicht nur den Wert von `"*c"`, sondern auch die drei folgenden Bytes, und da wir nicht wissen, ob diese drei Bytes irgendeine besondere Bedeutung haben, ist das Ergebnis völlig unbestimmt!

Es gibt aber einen besonderen Pointertypen, der für alle anderen stehen darf, nämlich `"void*"`. Das ist, wie der Name schon sagt, ein Zeiger auf „nichts Besonderes". Ein `"void"`-Pointer repräsen-

tiert zwar eine Speicheradresse, zeigt aber auf kein wirkliches Objekt, bzw. auf ein Objekt des Typs `"void"`.

```
void main()
{ int i;
  int *ip = &i;
  void *vp;

  vp = ip;        // Erlaubt
  ip = vp;        // In C++ falsch
  *vp = 99;       // Noch falscher
}
```

Der Zeiger `"ip"` darf ohne explizite Typwandlung an den Voidpointer `"vp"` zugewiesen werden, denn mit einem `"void"` kann man so gut wie nichts machen, so daß für das von `"ip"` referierte Datenobjekt keine Gefahr besteht. Umgekehrt wird aber bei der Zuweisung `"ip = vp"` ein Fehler gemeldet, denn hier wird ja wieder einmal ein „Zeiger auf irgendwas" in einen ganz bestimmten Zeigertypen umgewandelt. Falls Sie das wirklich wollen, müssen Sie hier wieder einen Cast setzen.

Die letzte Zeile zeigt dann auch, warum `"vp = ip"` zulässig war. Es ist nicht möglich, mit der Speicheradresse, auf die ein `"void*"` zeigt, etwas Gravierendes anzustellen, es sei denn, man wandelt den `"void"`-Pointer wieder in einen anderen Zeigertyp um, z. B. so:

```
*(int*)vp = 99;     // wandle "vp" erst in ein "int*" um
```

Nebenbei bemerkt haben wir es hier wieder mit einem Unterschied zwischen C und C++ zu tun: In C ist es nämlich erlaubt, ein `"void*"` ohne Casting in einen beliebigen Zeigertypen umzuwandeln. Das ist wieder einmal ein Beispiel dafür, daß man in C++ viel sauberer und sicherer programmieren kann als in C.

Man kann einen Zeiger auch, wenn es unbedingt sein muß, in eine Zahl umwandeln und wieder zurück. Ein Beispiel:

```
#include <stream.h>

void main()
{ int var;
  cout « "var liegt an Adresse: " « (unsigned int)&var;
}
```

Das obige Beispiel ist übrigens nicht portabel! Es ist keineswegs garantiert, daß ein `"int"` und ein Zeiger gleich breit sind (in diesem Fall: beide jeweils 32 Bit haben). Auf manchen Systemen ist ein `"int"` 16 Bit breit, wie `"short int"` unter MaxonC++, so daß im Beispiel nicht die Adresse, sondern nur deren untere 16 Bit ausgegeben würde. Wenn Sie in dieser Situation nach `"long"` wandeln, dürfte es auf den meisten Systemen klappen, aber noch nicht einmal das ist strenggenommen garantiert. Somit dürfte klar sein, daß Umwandlungen zwischen Zahlen und Zeigern eine etwas problematische Sache sind.

Eine Ausnahme gibt es allerdings: Die konstante Zahl `"0"` kann direkt in jeden Zeigertyp umgewandelt werden. Das benutzt man gern, um anzuzeigen, daß der Zeiger gerade auf „nichts" zeigt.

2.1.1.5 Was man sonst noch so mit Zeigern machen kann

Man darf Zeigerausdrücke gleichen Typs auch mit den Operatoren

```
==   !=   <   >   <=   >=
```

vergleichen. Dabei werden dann die Speicheradressen verglichen, als wären sie `"unsigned"`-Zahlen. So kann man mit `"=="` bzw. `"!="` feststellen, ob zwei Pointer auf dasselbe Objekt zeigen. Die anderen Operationen sind eigentlich nur dann sinnvoll, wenn die beiden Zeigerausdrücke auf Elemente desselben Vektors zeigen (siehe Abschnitt 2.1.2).

Natürlich kann man Zeiger auch mit dem ausgezeichneten Zeigerwert `"0"` vergleichen:

```
double *dptr;

// ...irgendwas

if (dptr == 0)
{ // usw.
```

Es wird garantiert, daß kein Datenobjekt an Adresse `0` liegt, so daß die `"if"`-Abfrage hier feststellt, ob `"dptr"` auf „nichts" zeigt.

Ein Zeigeraudruck kann auch direkt als logische Bedingung dienen und gilt als „wahr", wenn er nicht `"0"` ist. Also kann man die `"if"`-Anweisung verkürzen zu:

```
if (dptr)
{ // usw.
```

Ebenso kann ein Zeigerausdruck direkt als Operand der logischen Operatoren `"!"`, `"&&"` und `"||"` benutzt werden.

Leider ist es möglich, auf die von einem „Nullzeiger" referierte Adresse zuzugreifen:

```
double *dp = 0;

*dp = 17.4; // VORSICHT!
```

Dieses Programmfragment dürfte den Amiga zum Absturz bringen, denn an den Adressen `0` bis `7`, die hier gnadenlos überschrieben werden, liegen durchaus wichtige Betriebssystemdaten. Solche Aktionen sind, genau wie andere Operationen auf gar nicht oder sinnlos initialisierten Zeigern, deshalb möglich,

1. weil der Compiler nicht schon beim Übersetzen feststellen kann, ob eine Zeigervariable an einer bestimmten Stelle einen sinnvollen Wert hat. Ein so simples Beispiel wie das letzte, bei dem direkt nacheinander ein Zeiger auf `0` gesetzt wird und dann das referierte Objekt manipuliert

wird, könnte ein schlauer Compiler zwar abfangen. Es ist aber (und das ist mathematisch beweisbar!) möglich, jeden noch so geschickt geschriebenen Compiler hinters Licht zu führen.

2. weil es einfach zu viel Rechenzeit kosten würde, zur Laufzeit des Programms vor jeder Zeigeroperation erst festzustellen, ob der Pointer auf einen sinnvollen Wert zeigt. Auch hier gilt der Grundsatz: „Der C-Programmierer weiß, was er tut."

Pointer haben vieles mit den Vektoren, die im nächsten Kapitel besprochen werden, gemeinsam. Deshalb kann man die binären Operatoren `"+"` und `"-"` sowie den Indexoperator `"[ ]"` darauf anwenden. Aber das gehört dann auch schon in das folgende Kapitel:

2.1.2 Vektoren

2.1.2.1 Das Vektor-Konzept in C

Praktisch jede Programmiersprache kennt so etwas wie Felder, Arrays oder Vektoren, nur die Bezeichnungen sind unterscheidlich und werden auch beliebig durcheinander verwendet. Das Prinzip ist aber immer das gleiche: Ein Datenobjekt besteht aus einer endlichen Anzahl von Elementen, die jeweils denselben Typ besitzen und durchnumeriert sind. Sowohl der Vektor als auch jedes einzelne seiner Elemente kann man als Datenobjekt auffassen.

Sei `"v"` ein Vektor, der aus 10 Elementen des Typs `"int"` besteht. Dann heißen die einzelnen Elemente

```
v[0], v[1], v[2], ..., v[8], v[9]
```

Also greift man auf ein Element eines Vektorausdrucks zu, indem man den „Index", der ein beliebig komplizierter `"int"`-Ausdruck sein kann, in eckigen Klammern hinter den Vektorausdruck stellt. Man beachte, daß Indizes (bitte nicht „Indexe" sagen!) in C immer von `0` an gezählt werden. Jedes dieser Vektorelemente ist dann eine eigenständige Variable (hier: ein `"int"`-Objekt) und kann auch als solche in Programmen verwendet werden.

Die eckigen Klammern gelten übrigens als Operator, und zwar als Postfix-Operator, der stärker bindet als z. B. die vorangestellten unären Operatoren.

Natürlich müssen auch Vektoren deklariert werden. Wieder einmal gilt hier, daß die Deklaration wie die Benutzung aussehen soll. Also wird der Vektor `"v"` so deklariert:

```
int v[10];  // deklariert den Vektor v[0], v[1], ..., v[9] !!!
```

Ein kleines Beispiel für Vektoren:

```
// Ein kleines Statistik-Programm

#include <stream.h>

void main()
```

```
{ int i;
  double dat[20];

  cout « "Bitte geben Sie 20 Zahlen ein:\n";

  for (i=0; i<20; i++)
  { cout « "Zahl " « i+1 « ": ";
    // C zählt von 0 bis 19, der Benutzer wahrscheinlich lieber
    // von 1 bis 20. Deshalb wird hier "i+1" ausgegeben.
    cin » dat[i];
  }

  double sum = 0;
  for (i=0; i<20; i++)       // Wie summieren die
    sum += dat[i];             // eingegebenen Zahlen

  double mittel = sum/20;    // Mittelwert = Summe/Anzahl

  cout « "\nSumme:       " « sum;
  cout « "\nMittelwert: " « mittel;
  cout « "\n";
}
```

Leider muß die Größe eines Vektorobjekts immer eine Konstante sein. Wenn das Programm dahingehend verbessert werden soll, daß der Benutzer wählen kann, wie viele Zahlen er einzugeben gedenkt, bleibt uns (vorerst) nichts anderes übrig, als einen ziemlich großen Vektor (vielleicht `"dat[1000]"`) einzurichten und dann nur die benötigten Elemente davon zu benutzen.

C prüft nicht, ob ein Index im „erlaubten" Bereich liegt! Hätte man oben nur `"dat[10]"` deklariert und dann trotzdem 20 Elemente benutzt, wäre das Ergebnis `"undefiniert"`, was in der C-Sprachdefinition der übliche Euphemismus für „Absturz" ist. Auch hier gilt einmal wieder: „Wer in C programmiert, muß wissen, was er tut".

Es gibt keine mehrdimensionalen Vektoren, sondern nur „Vektoren von Vektoren", was im Prinzip auf dasselbe hinauskommt. Ein Schachbrett, das bekanntlich aus 8*8 Feldern besteht, könnte man so darstellen:

```
char Brett[8][8];
```

Entsprechend kann man dann mit `"Brett[ i ][ j ]"` auf das Element in der `"i"`-ten Zeile und `"j"`-ten Spalte zugreifen, oder umgekehrt. Das hängt ganz davon ab, ob Sie den ersten Index nun als Zeilen- oder Spaltennummer betrachten. Andererseits stellt dann aber auch `"Brett[i]"` eine einzelne Zeile (bzw. Spalte) dar. Wir haben es also tatsächlich mit einem Vektor von Vektoren und nicht einem einzelnen zweidimensionalen Vektor zu tun.

2.1.2.2 Zeiger und Vektoren

Jetzt werden Sie sich vielleicht wundern, aber ein Vektor und ein Zeiger sind in C fast dasselbe! Dies gehört zu jenen komischen Features, die den Leuten damals bei der Einführung von C ungeheuer toll vorkamen, heute aber nur noch ärgerliche Anachronismen sind.

Die Sache ist, kurz gesagt, folgende: Ein Pointer verweist normalerweise auf eine Speicherstelle. Aber oft sind ja auch die Speicherbereiche davor und dahinter von Interesse. Deshalb kann man einen Zeiger ebenfalls mit einem Index versehen, als ob er ein Vektor wäre. Betrachten wir einmal folgende Deklarationen:

```
int vector[100];
int *pointer = &vector[20];
```

Nun zeigt `"pointer"` auf das zwanzigste Element von `"vector"`, und wir können `"*pointer"` synonym zu `"vector[20]"` benutzen. Intern wird ein Vektor aber realisiert, indem die Elemente hintereinander im Speicher liegen. Direkt hinter `"vector[0]"` liegt im Speicher also `"vector[1]"`, worauf `"vector[2]"` folgt usw. Jetzt machen wir eine kleine Wertzuweisung, die Sie vielleicht verwundern wird:

```
pointer = pointer+5;
```

Wird zu einem Zeigerausdruck ein ganzzahliger Ausdruck addiert, so ist das Ergebnis ein Zeiger auf die um entsprechend viel Vektorelemente weiter liegende Speicheradresse. Also ist `"pointer+5"`, wenn `"pointer"` auf das 20-ste Element von `"vector"` zeigt, ein Pointer auf `"vector[25]"`. Man kann im Prinzip zu jedem Zeigerausdruck etwas addieren, aber Sinn hat das natürlich nur, wenn der Zeiger gerade in einen Vector zeigt.

Entsprechend kann man auch die Subtraktion `"-"` und die Operatoren `"+="`, `"-="`, `"++"` und `"--"` auf Zeiger anwenden: Nach

```
pointer -= 5;
```

zeigt unser `"pointer"` wieder auf das zwanzigste Element von `"vector"`.

Nun dürfte klar sein, daß

```
*(pointer+n)
```

den Inhalt des Vektorelements liefert, der `"n"` Positionen hinter dem liegt, auf das `"pointer"` gerade zeigt. Und dafür gibt es (die Überraschungen reißen nicht ab) eine schönere Schreibweise:

```
pointer[n]
```

Das sieht jetzt aus wie ein ganz normaler Vektorzugriff, und das ist es auch. Wieso das?

Computer können von Natur aus leicht mit Zeigern, die ja im Prinzip nur eine Speicheradresse repräsentieren, umgehen. Mit so großen und unhandlichen Gebilden wie Vektoren tun sie sich dagegen relativ schwer, was ganz einfach daher kommt, daß ein so großes Datenobjekt nicht in ein Prozessorregister paßt, sondern langsam und umständlich durch den Speicher geschoben werden muß. Also entschied man bei der Definition von C, Vektoren stets nur als Zeiger zu verwenden: Nach der Deklaration

```
double dat[10];
```

wird für "dat" zwar ein zehn Elemente umfassender "double"-Vektor eingerichtet. Der Bezeichner "dat" ist dann aber ein Ausdruck des Typs „Zeiger auf double" und entspricht einem Zeiger auf das erste (also „nullte") Vektorelement:

dat ist identisch mit &dat[0]

Folglich könnte man schreiben:

```
int vector[100];
int *pointer = vector;     // Entspricht "&vector[0]"
```

Um "pointer" wieder auf das zwanzigste Element zu setzen, kann man also

```
int vector[100];
int *pointer = vector+20;
```

deklarieren. Der „normale" Vektorzugriff

```
v[i]
```

ist nur eine andere Schreibweise für

```
*(v+i)
```

Mit diesen Features kann der echte C-Freak zwar echt lässig durch die Gegend pointern, aber übersichtlich ist das natürlich nicht.

An dieser Stelle ist wohl wieder ein Beispiel angebracht: Das folgende Beispiel setzt alle Elemente eines "int"-Vektors auf Null, benutzt dafür aber keine Schleife von 0 bis 99, sondern läßt einen Zeiger vom 0-ten bis zum 99-sten Element laufen:

```
void main()
{ int vector[100];
  for(int *pointer = vector; pointer != &vector[100];
    pointer++)
    *pointer = 0;
}
```

Der Zeiger "pointer" wird also zu Beginn der "for"-Schleife auf das erste Vektorelement gesetzt. Solange "pointer" nicht auf das hundertste Element zeigt (das es eigentlich gar nicht gibt, da der Vektor nur bis 99 reicht), wird das von "pointer" referierte Element auf 0 gesetzt und anschließend der Zeiger inkrementiert, also um ein Element weitergesetzt.

Nervenstarken C-Anfängern sei noch gesagt, daß ein ultimativer Hard-Core-C-Coder folgendes geschrieben hätte:

```
void main()
{ int vector[100], *pointer = vector+100;
  while(pointer > vector)
    *--pointer = 0;
}
```

```
/* Der Zeiger "pointer" wird mit der Endadresse des Vektors "vector"
initialisiert. Solange er größer ist als die Anfangsadresse von
"vector", wird er dekrementiert, d. h. um ein Vektorelement
zurückgesetzt, und dann das referierte Vektorelement auf 0 gesetzt */
```

Einer der vielen Schwachpunkte des Vektorkonzepts in C besteht darin, daß man Vektoren nicht einander zuweisen kann. Folgendes ist z. B. nicht möglich:

```
void main()
{  int v1[50], v2[50];   // Zwei Vektoren

   v1 = v2;              // ERROR
}
```

Das folgt direkt aus diesem seltsamen Vektorkonzept, denn `"v1"` ist demzufolge ja nur die Anfangsadresse der ersten Vektorelements, und eine Adresse allein ist kein L-Wert. Somit wäre die Zeile

```
v1 = v2;  // Gedacht als Wertzuweisung zwischen Vektoren
```

gleichbedeutend mit

```
&v1[0] = &v2[0]; // Gleichsetzen von Adressen? Häh???
```

und das hat ja nun wirklich keinen Sinn. Wenn man in C einen Vektor in einen anderen kopieren will, muß man das (am besten mit einer Schleife) elementweise tun.

2.1.2.3 Genaueres über Vektordeklarationen

Bisher haben wir Vektoren immer in der Form

```
Grundtyp Variablenname [ Anzahl ]
```

deklariert. Eine gültige Typbeschreibung erhält man in C immer, indem man bei einer Variablendeklaration den Variablennamen wegläßt. Also ist

```
int [10]
```

die korrekte Beschreibung des Datentyps „int-Vektor mit zehn Elementen".

Natürlich kann man Zeiger- und Vektortypen miteinander verbinden. Dann kann es aber zu etwas kryptischen Deklarationen kommen, z. B.

```
char *komisch[20];
```

Was ist das jetzt, ein Vektor von Zeigern oder ein Zeiger auf einen Vektor? Hier helfen wieder die beiden goldenen Regeln für das Überleben verworrener C-Deklarationen:

1. Typen und Deklarationen liest man von innen nach außen.
2. Die Deklaration eines Objekts soll idiotischerweise wie die Benutzung aussehen.

Nach Regel 2 könnten wir obiges Objekt beispielsweise so benutzen:

```
*komisch[i]
```

(wobei das Ergebnis dann, wie die Deklaration ja sagt, ein `"char"` ist - wenigstens das wissen wir ja). Nach Regel 1 fangen wir bei der Interpretation innen (beim Bezeichner `"komisch"`) an und müssen nur noch herausfinden, ob wir jetzt zuerst den Index oder das `"*"` anwenden. Wenn Sie gut aufgepaßt haben, wissen Sie noch, daß ein Index als Postfix-Operator gilt, daß `"*"` unter die Rubrik „unärer Operator" fällt und daß Postfix-Operatoren stärker binden als unäre Operatoren. Folglich wird zuerst `"[i]"` auf `"komisch"` angewendet und erst dann `"*"` auf das Ergebnis. Ergo ist `"komisch"` ein Vektor von Zeigern auf `"char"`.

Um einen Zeiger auf einen Vektor zu deklarieren, müssen Sie also Klammern setzen:

```
char (*sehrkomisch)[20];
```

Dann liefert

```
(*sehrkomisch)[i]
```

das `"i"`-te Zeichen in dem `"char"`-Vektor, auf den `"sehrkomisch"` zeigt. Ganz schön komisch, was?

So, und spätestens jetzt dürfte Ihnen wohl klar sein, daß es in der C-Syntax immer noch etwas schlimmer kommen kann, als man denkt. Also dürfte es Sie auch nicht weiter schockieren, daß man die Kombination von Zeiger- und Vektortypen beliebig weit treiben kann, etwa

```
int *(*(**x)[2])[3]
```

Das ist dann - immer schön nach unseren beiden Regeln - ein Zeiger auf einen Zeiger auf einen Vektor zweier Zeiger auf je einen dreielementigen Vektor von Zeigern auf int. Das einzig Schöne an solchen Datentypen ist, daß man sie normalerweise nicht braucht und sich deshalb nicht darum kümmern muß.

Dagegen tauchen Datentypen wie

```
int (*)[10]
```

tatsächlich bisweilen auf (z. B. in Parameterlisten) und bringen auch erfahrene C-Programmierer in Verlegenheit. Die Klammern um das Pointersymbol `"*"` sehen doch ziemlich seltsam aus, zumal

```
int *[10]
```

etwas ganz anderes ist! Hier sollten Sie immer daran denken, daß Datentypen in C dummerweise wie Variablendeklarationen aussehen, nur daß man eben den Variablennamen wegläßt, und der müßte hier in beiden Fällen direkt hinter dem `"*"` stehen. Also ist ersteres die korrekte Schreibweise für „Zeiger auf Vektor", während letzteres „Vektor von Zeigern" heißt.

2.1.2.4 Vektoren als Parameter

Oft will man einen Vektor als Argument an eine Funktion übergeben. Leider kann C auch das nicht, jedenfalls nicht so richtig: das geht nur über Pointer. Nehmen wir einmal an, Sie deklarieren folgende Funktion:

```
void Initialisiere (int v[])
{ for (int i=0; i<10; i++)
     v[i] = 0;
}

void main()
{ int vector[10];
  Initialisiere(vector);
}
```

Wenn man einen skalaren Datentypen - also numerisch oder Pointer - als Argument übergibt, wird bekanntlich eine Kopie des Argumentwerts auf dem Stack abgelegt. Bei Vektor-Parametern macht C aber Probleme: Bei der Funktionsdeklaration ersetzt der Compiler automatisch `"int[10]"` durch `"int*"`. Der Parameter `"v"` ist dadurch also kein Vektor, sondern bloß ein Zeiger auf einen solchen. Es hätte auf das Programm keinerlei Einfluß gehabt, wenn man

```
void Initialisiere (int *v)
```

deklariert hätte.

Daran erkennen Sie unschwer, daß es keine rechte Bedeutung hat, ob wir als Argument nun einen Vektor mit 10 oder 100 Elementen übergeben. Deshalb ist bei Parameterdeklarationen auch die folgende Schreibweise erlaubt:

```
void Initialisiere (int v[])
```

Hier sieht man dann sofort, daß prinzipiell ein beliebiger `"int"`-Vektor als Argument verwendet werden könnte. Wenn der dann allerdings weniger als zehn Elemente hat, schreibt die Funktion gnadenlos darüber hinaus und bringt den Recher schätzungsweise zum Absturz. Deshalb ist es in solchen Situationen klüger, die Vektorgröße als zusätzlichen Parameter zu erwarten, etwa so:

```
void Initialisiere (int v[], int n)
{ for (int i=0; i<n; i++)
     v[i] = 0;
}

void main()
{ int vector[10];
  Initialisiere(vector, 10);
}
```

Mehrdimensionale Vektoren gibt es nicht, nur Vektoren von Vektoren. Erwartet eine Funktion ein derartiges Argument, etwa

```
void Ausgabe (char c[10][80]);
```

so wird in Wirklichkeit ein Zeiger auf `"char[80]"` übergeben. Also wären folgende Deklarationen äquivalent:

```
void Ausgabe (char c[10][80]);
void Ausgabe (char c[ ][80]);
void Ausgabe (char (*c)[80]);
```

Syntaktisch falsch sind dagegen folgende Deklarationen:

```
void Ausgabe (char c[ ][ ]);
void Ausgabe (char c[10][ ]);
```

In beiden Fällen weiß der Compiler zwar, daß der Parameter ein Zeiger auf einen Vektor ist, aber nicht, wie viele Elemente dieser Vektor hat. Also kann er damit nichts Rechtes anfangen, und er beschwert sich.

2.1.2.5 Strings

Konstante Zeichenketten

Bisher hatten wir Zeichenketten nur in konstanter Form kennengelernt. Ein konstanter String ist aber nichts anderes als ein Vektor von Zeichen. Damit man das Ende einer Zeichenkette feststellen kann, wenn man ihre Länge gerade nicht kennt, steht in C am Ende einer Zeichenkette immer ein Nullbyte, wodurch ein String immer um ein Element verlängert wird. Deshalb hat der Ausdruck

```
"Hallo"
```

den Typ

```
char[6]
```

Man kann eine konstante Zeichenkette sogar mit einem Index versehen:

```
cout « "Hallo"[1];
```

gibt a aus.

Eine konstante Zeichenkette kann an einen `"char"`-Zeiger zugewiesen werden, denn allgemein sind in C Zuweisungen `"T* = T[ ]"` für beliebige Typen `"T"` erlaubt. Ein Beispiel:

```
char *string1;

string1 = "Mach's gut, und danke für den Fisch!";
```

Natürlich zeigt `"string1"` jetzt auf die entsprechende konstante Zeichenkette und enthält nicht etwa eine Kopie davon. Deshalb ist es zwar erlaubt, aber eine ziemlich haarige Angelegenheit, eine Stringkonstante zu verändern. Auch dazu ein Beispiel:

```
char *st1 = "Hallo";  st1[1] = 'e';
```

Nun sollte die konstante Zeichenkette von `"Hallo"` in `"Hello"` verwandelt worden sein, was natürlich ziemlich komisch ist, denn auf diese Art und Weise ist eine konstante Zeichenkette ja alles

andere als konstant. Deshalb wird auch allgemein von solchen Konstruktionen abgeraten. Noch dicker kommt es, wenn der Compiler Zeichenketten optimiert:

```
char *st1 = "Test", *st2 = "Test";
st1[2] = 'x';          // Ändere st1 von "Test" nach "Text"
```

Zunächst zeigen beide Pointer auf eine Konstante namens `"Test"`, aber der ANSI-Standard legt nicht fest, ob es sich dabei um dieselbe Konstante handelt! MaxonC++ legt hier zwei Kopien des Strings an, also zeigt `"st2"` nachher immer noch auf `"Test"`. Ein anderer Compiler könnte aber durchaus nur einmal die Konstante `"Test"` anlegen, so daß `"st1"` und `"st2"` auf ein und denselben Speicherbereich zeigen, mit der Folge, daß über `"st1"` immer auch `"st2"` modifiziert wird. Also Pfoten weg von Schreibzugriffen auf konstante Zeichenketten!

Initialisieren, Kopieren und Zuweisen

Ein Vektor von `"char"`-Elementen kann mit einer konstanten Zeichenkette initialisiert werden:

```
char c[20] = "Hallo";
```

Nach dieser Initialisierung gilt:

```
c[0] == 'H', c[1] == 'a', ..., c[4] == 'o', c[5] == '\0'
```

Die restlichen Elemente von `"c"` sind undefiniert.

Bei einer solchen Initialisierung muß der String einschließlich des Nullbytes am Ende in die Variable hineinpassen, sonst meldet der Compiler einen Fehler. Dabei gibt es wieder einen jener spitzfindigen Unterschiede zwischen C und C++: In ANSI C wird bei solchen Initialisierungen nicht verlangt, daß das Nullbyte in die Zeichenkette passen muß. Also ist folgendes unter MaxonC++ nur im C-Modus möglich:

```
#pragma -
char s[4] = "Test";   // 4 Zeichen + Nullbyte: nur in C erlaubt!
```

Wenn man diesen String dann aber ausgeben läßt, kann es sein, daß man eine Überraschung erlebt, denn dabei wird stets alles vom Stringanfang bis zum ersten Nullbyte ausgegeben. Da `"s"` selbst hier kein solches Endzeichen besitzt, wird auch noch das ausgegeben, was zufällig gerade im Speicher hinter `"s"` liegt.

Strings sind elementare, nahezu unverzichtbare Datentypen. Daher ist es schade, daß C sie kaum unterstützt. Wertzuweisungen zwischen `"char"`-Vektoren sind, genau wie bei allen anderen Vektoren, nicht direkt möglich:

```
#include <stream.h>

void main()
{ char str1[20] = "Herforder Pils", str2[20];
  str2 = str1;      // ERROR
  cout « str2;
}
```

Hier bleibt uns nur eins übrig: Wir müssen die Strings Zeichen für Zeichen kopieren. Damit das nicht zu mühsam wird, schreiben wir dazu eine kleine Funktion namens `"strcpy"` wie „String-Copy". Erfreulicherweise wird ein String immer mit einem Nullzeichen abgeschlossen. Deshalb braucht `"strcpy"` nicht zu wissen, wie lang der zu kopierende String ist, sondern kann das selbst feststellen. Auf diese Weise kann man dann `"str2 = str1"` einfach durch

```
strcpy(str2, str1)
```

ersetzen.

Ein erster Entwurf von `"strcpy"` könnte so aussehen:

```
void strcpy(char ziel[ ], char quell[ ])
{ int i;
  for (i=0; quell[i] != '\0'; ++i) // Zeichenweise von "quell"
  ziel[i] = quell[i];  // nach "ziel" kopieren
  ziel[i] = '\0';          // Nullbyte ans Ende von "ziel"
}
```

Das ist eine übersichtliche, gut lesbare Version von `"strcpy"`, wie sie ein C-Anfänger schreiben würde. Ein Fortgeschrittener würde dagegen folgendes tun:

1. Man deklariert die Parameter gleich als Pointer, denn der Compiler wandelt Vektor-Parameter ja sowieso dahin um.
2. Man kann `"quell[i]"` direkt als Bedingung verwenden, denn das bedeutet ja nichts anderes als `"quell[i] != 0"`.
3. Wenn man direkt mit Pointern operiert, spart man sich die Indexvariable.
4. Die Konstante `"'\0'"` kann man ganz einfach durch `"0"` ersetzen, denn in C sind Zeichen ja nichts anderes als Zahlen.

Eine fortgeschrittenere Version von `"strcpy"` sähe dann etwa so aus:

```
void strcpy(char *ziel, char *quell)
{ while (*quell)// Solange "quell" auf ein Zeichen != 0 zeigt, ...
  *ziel++ = *quell++;   // wird dieses nach "*ziel" kopiert,
                                // und beide Zeiger werden
                    // um je ein Zeichen weitergesetzt.
  *ziel = 0;  // Nullbyte ans Ende des Zielstrings schreiben.
}
```

Bitte erzählen Sie mir jetzt nicht, daß das ganz schön unübersichtlich ist - ich bin mir darüber im klaren. Aber erstens ist diese Version kürzer und zweitens schneller als die alte.

In der Praxis brauchen Sie sich zum Glück nicht den Kopf zu zerbrechen, wie Sie `"strcpy"` nun implementieren, denn diese Funktion gibt es bereits fix und fertig. Sie wird im Includefile `"<string.h>"` deklariert und bei Bedarf vom Linker eingebunden:

```
#include <stream.h>
#include <string.h>
```

```
void main()
{ char str1[20] = "Herforder Pils", str2[20];
  strcpy (str2, str1);
  cout « str2;
}
```

Bitte denken Sie daran, daß `"strcpy"` eine Zeichenkette gandenlos kopiert, ganz egal, ob sie nun in die Zielvariable hineinpaßt oder nicht. Unter Umständen wird dann über die Stringvariable hinausgeschrieben, und das hat meist fatale Folgen. Etwas sicherer ist deshalb die Funktion `"strncpy"`, die als zusätzliches Argument die maximale Anzahl der zu kopierenden Strings erhält. Dabei wird aber nicht automatisch ein Nullbyte an das Stringende gesetzt, sondern nur dann, wenn der zu kopierende String auch tatsächlich in den Zielbereich hineinpaßt.

Ein Beispiel dafür:

```
#include <stream.h>
#include <string.h>

void main()
{ char s1[40] = "Live fast - love hard - die young.",

    // J. Joplin
  s2[40] = "On the sunny side of the street.";       // Cadillac

  strncpy(s2, s1, 12);    // maximal 12 Zeichen kopieren

  cout « s2;          // Ergebnis: "Live fast -  side of the street."
}
```

Diesen Effekt kann man auch ausnutzen, um gezielt in einen String hinein zu kopieren. Auch dazu ein Beispiel:

```
#include <stream.h>
#include <string.h>

void main()
{ char s1[20] = "Piss off!";

  strncpy(s1+1, "***", 3);     // String wird zensiert

  cout « s1;          // Ausgabe: "P*** off!"
}
```

Hätte man hier `"strcpy"` statt `"strncpy"` benutzt, wäre auch das vierte Zeichen der Konstanten `"***"`, nämlich das Nullbyte, kopiert und so `"s1"` dahinter abgeschnitten worden.

Stringvergleiche

Oft will man den Inhalt von Strings vergleichen, etwa um festzustellen, ob sie gleich sind oder ob ein String alphabetisch vor dem anderen steht. Dann geht das Elend mit den C-Strings schon wieder los: Wenn man Strings, also Vektoren von Zeichen, mit den normalen Operatoren wie `"=="` oder

"<" vergleicht, vergleicht man effektiv Zeiger auf Zeichen und somit die Anfangsadressen der Strings. Ein Beispiel:

```
void main()
{ char s1[10] = "Test",
  s2[10] = "Test",
  *s3 = s1,
  *s4 = "Test"; // Es gilt: s1 == s3, aber s1 != s2 und s1 != s4
}
```

Hier wird die Zeichenkette "Test" in die Stringvariablen "s1" und "s2" kopiert, und "s4" zeigt auf eine weitere Kopie dieses Strings, während "s3" auf den String "s1" verweist. Obwohl alle drei Variablen bei der Ausgabe (z. B. "cout « s1";) "Test" liefern, sind nur "s1" und "s3" gleich (weil die Adresse übereinstimmt), während alle anderen Variablen voneinander verschieden sind.

Derartige Vergleiche sind manchmal wirklich sinnvoll, aber meist will man den tatsächlichen Inhalt von Strings vergleichen und nicht ihre Speicheradressen. Auch hier hilft uns eine Bibliotheksfunktion aus "<string.h>" aus der Klemme: Sie heißt "strcmp" und erhält als Parameter zwei Zeichenketten (das heißt in C immer: Zeiger darauf).

```
"strcmp(a,b)" ist: = 0, wenn "a" und "b" gleich sind
   < 0, wenn "a" im Alphabet vor "b" steht
   > 0, wenn "a" nach "b" kommt.
```

„Alphabetische Reihenfolge" bezieht sich hier auf den ASCII-Code. Es werden zunächst die jeweils ersten Zeichen der Strings verglichen. Sind diese gleich, werden die jeweils folgenden Zeichen verglichen und so weiter. Die Strings werden höchstens bis zum Nullbyte am Ende verglichen. Dieses Verfahren liefert im allgemeinen nicht die Reihenfolge, die man sich so landläufig unter „alphabetisch" vorstellt. Insbesondere kommen für den Computer alle Großbuchstaben vor sämtlichen Kleinbuchstaben.

Mit

```
if (strcmp(x,y) == 0)
```

kann man also feststellen, ob der Inhalt der Zeichenketten "x" und "y" gleich ist. Natürlich kann man das auch verkürzen zu

```
if (!strcmp(x,y))
```

Um ein bißchen den Umgang mit Strings zu üben, wollen wir hier die Funktion "strcmp" einmal selbst schreiben. Ein erster, simpler Versuch wäre

```
int strcpy(unsigned char s1[ ], unsigned char s2[ ])
{ int i = 0;

   // Wir übergehen alle identischen Zeichen
   // am Anfang der Strings:
   while (s1[i] != '\0' && s2[i] != '\0' && s1[i] == s2[i])
      i++;
```

```
  // Ergebnis auswerten:
  if (s1[i] < s2[i])
    return -1;
  else
    if (s1[i] > s2[i])
      return +1;
    else
       return 0;
}
```

Bei `"strcpy("Test", "Text")"` wird hier `"i"` zweimal heraufgezählt, denn die beiden ersten Zeichen der beiden Zeichenketten sind ja identisch. Dann werden die Zeichen `'s'` und `'x'` verglichen, und als Ergebnis wird `"-1"` zurückgegeben, denn `'s'` ist kleiner als `'x'`. Hilfreich ist hier, daß das Nullzeichen, das ja das Stringende markiert, bei vorzeichenlosen `"char"`'s kleiner als alle anderen Zeichen ist. Deshalb kann man, wenn man mit `"unsigned char"`-Werten arbeitet, relativ problemlos den Fall, daß bei einem String das Stringende erreicht wird, abhandeln. Beim Vergleich von `"Test"` mit `"Teststring"` werden vier Zeichen übersprungen und dann das Nullzeichen am Ende von `"Test"` mit dem zweiten `'s'` aus `"Teststring"` verglichen. Das Nullbyte ist natürlich kleiner als das `'s'` und also `" Test"` kleiner als `"Teststring"`, wie man das auch nicht anders erwartet.

Optimal (im Sinne eines „harten" C-Programmierers) ist die Funktion aber noch lange nicht. Damit uns niemand „Lamer" nennt, ändern wir umgehend einige Details der Funktion:

1. Die Parametertypen ändern wir in `"unsigned char*"` um, denn der Compiler tut das ja sowieso. Außerdem lassen wir das `"unsigned"` wegfallen und sehen später, wie wir damit zurecht kommen.
2. Wenn wir schon mit Zeigern arbeiten, dann bitte auch richtig. Also fällt auch das `"i"` flach, und wir setzen direkt die Zeiger weiter.
3. Niemand verlangt von uns, daß wir bei unterschiedlichen Strings gerade `"-1"` oder `"+1"` zurückgeben, es muß eben nur das Vorzeichen stimmen. Deshalb können wir die beiden Vergleichsoperationen durch eine simple Subtraktion ersetzen.

Mit diesen Änderungen sieht das Ganze so aus:

```
int strcpy(char *s1, char *s2) // Zweite Version
{  // Wir übergehen alle identischen Zeichen
   // am Anfang der Strings:
   while (*s1 != '\0' && *s2 != '\0' && *s1 == *s2)
   {  ++s1;
      ++s2;
   }

   // Ergebnis auswerten:
   return (unsigned char)*s1 - (unsigned char)*s2;
}
```

Wir suchen also wieder das erste Zeichen, in dem sich "s1" und "s2" unterscheiden. Diese Zeichen wandeln wir dann in "unsigned char" um und berechnen die Differenz. Das geschieht übrigens den Sprachregeln von ANSI C entsprechend im Typ "int", so daß wir uns keine Sorgen wegen Überläufen machen müssen.

Die Funktion ist jetzt schon viel kürzer geworden, aber zwei Details stören immer noch: Zum einen sind die Vergleiche "!= '\0'" redundant, da C ja nicht zwischen numerischen und logischen Datentypen unterscheidet, zum anderen kann man einen der beiden ersten Vergleiche weglassen, wie im folgenden geschehen:

```
int strcpy(char *s1, char *s2) // Dritte Version
{ while (*s1 && *s1 == *s2)
  { ++s1;
    ++s2;
  }

  return (unsigned char)*s1 - (unsigned char)*s2;
}
```

Für alle, die daran zweifeln, daß man das "*s2 != '\0'" wirklich weglassen kann, sei gesagt, daß es vier verschiedene Fälle gibt:

- Die Strings sind gleich. Dann erreichen wir mit beiden Zeigern gleichzeitig das Stringende und brechen die Schleife aufgrund der Klausel "*s1" ab.
- Die Strings unterscheiden sich schon vor beiden Stringenden. Dann stoßen wir irgendwann auf eine Stelle, wo "*s1 == *s2" nicht mehr erfüllt ist, und die Schleife endet wiederum wunschgemäß.
- Wir erreichen zuerst das Ende von "s1", da beide Strings bis hierhin identisch sind. Dann ist "*s1" falsch, und die Schleife bricht ab.
- Wir erreichen zuerst das Ende von "s2". Dann ist zwar "*s1" noch erfüllt, aber "*s1 == *s2" gilt natürlich nicht mehr, denn s2 zeigt auf ein Nullbyte und s1 nicht.

Die dritte Version von "strcmp" ist wieder einmal ein schönes Beispiel dafür, wie man in C äußerst kurze und schnelle Programme schreiben kann, die dann aber leider niemand mehr ohne weiteres versteht.

Am Ende dieses Abschnitts sei noch die Funktion "strlen" erwähnt, die ebenfalls in "<string.h>" deklariert wird. Sie liefert die Länge einer Zeichenkette, und zwar ausschließlich des Nullbytes am Ende. Spaßeshalber versuchen wir auch hier eine eigene Implementierung. Zuerst wieder eine einfache, verständliche Version:

```
int strlen(char *s)   // Erste Version
{ int i = 0;
  while (s[i] != '\0')
    ++i;
  return i;}
```

Das optimieren wir wieder, indem wir von Index- auf Zeigeroperationen umsteigen. Wir müssen uns dabei den Stringanfang merken und einen zweiten Zeiger bis zum Ende wandern lassen. Am Ende subtrahieren wir dann die beiden Pointer und erhalten so die Stringlänge (Sie erinnern sich: Die Differenz zweier Zeiger gleichen Typs ist die Anzahl der Vektorelemente, die dazwischen paßt):

```
int strlen(char *s)   // Zweite Version
{ char *t = s;
  while (*t)
    ++t;
  return t-s;
}
```

Das ist dann in der Tat eine Funktion, die sich wohl kaum weiter optimieren läßt - es sei denn, Sie verwenden die Bibliotheksfunktion `"strlen"` von MaxonC++, denn die ist in Assembler geschrieben und deshalb noch etwas kürzer und schneller.

Argumente aus der Kommandozeile

Weil Sie gerade Strings kennengelernt haben und es sonst nirgendwo sinnvoll in dieses Handbuch paßt, verrate ich Ihnen hier, was es mit `"argc"` und `"argv"` auf sich hat:

Wenn man auf dem Amiga ein Programm von CLI startet, kann man hinter dem Programmnamen bekanntlich noch Argumente angeben. Natürlich haben diese Argumente meist einen tieferen Sinn, und folglich muß das Programm, das z. B. in C++ geschrieben ist, diese auch auswerten.

Für diesen Zweck gibt es in C eine standardisierte Lösung, die Argumente werden als Parameter an das Hauptprogramm, also `"main"`, übergeben. Wenn man sie haben will, muß man folgende Parameter für `"main"` deklarieren:

```
void main(int argc, char *argv[ ]);
```

Der erste Parameter, der in der Regel `"argc"` genannt wird, enthält die Anzahl der Argumente, der zweite ist ein Vektor mit Zeigern auf eben diese, wobei der Programmname ebenfalls als Argument zählt. Vor dem Start des Programms wird nämlich die Argumentzeile in Worte zerlegt:

```
#include <stream.h>

 void main(int argc, char *argv[])
{ int i;
  for (i=0; i<argc; ++i)
    cout « "Parameter " « i « ": " « argv[i] « "\n";
}
```

Angenommen, Sie speichern das ausführbare Programm unter dem Namen `"test"` ab und geben im CLI

```
test Wer bin ich.
```

ein. Dann hat der Parameter `"argc"` den Wert 4, und die vier ersten Zeiger in `"argv"` zeigen auf die Strings `"test"`, `"Wer"`, `"bin"` und `"ich."`. Natürlich müssen Sie aus der integrierten Umge-

bung von MaxonC++ nicht extra ein Exe-File abspeichern, sondern können die Parameterübergabe mit dem Texteingabefeld ***„Parameter"*** aus dem Menü **„Optionen"** simulieren. Als Programmname wird dabei natürlich nichts allzu Sinnvolles übergeben, den das Programm existiert ja nicht als ausführbare Programmdatei.

2.1.2.6 Initialisierung von Vektoren

Vektoren sind die ersten Datentypen, bei denen es einen Unterschied zwischen Initialisierung und Wertzuweisung gibt. Eine wirkliche Wertzuweisung ist nur elementweise möglich, aber initialisieren darf man sie. Sie haben bereits gesehen, daß man Stringvariablen mit einer konstanten Zeichenkette oder einem anderen Stringausdruck initialisieren darf, während man für eine entsprechende Wertzuweisung auf die `"strcpy"`-Funktion zurückgreifen müßte. Ganz allgemein sind für Vektoren zwei verschiedene Initialisierungen erlaubt:

1. Ein Vektorausdruck desselben Elementtyps. Die Anzahl der Elemente muß dabei nicht übereinstimmen. Hat der Ausdruck mehr Elemente als die Variable, werden die überzähligen ignoriert, und wenn es zu wenige sind, bleiben entsprechend viele Vektorelemente undefiniert.

Beispiel:

```
void main()
{ int v1[10];
  for (int i=0; i<10; i++)      // v1 wird initialisiert
    v1[i] = i+42;

  int v2[15] = v1; // initialisiert v2 mit v1, wobei v2[10]
                   // bis v2[14] undefiniert bleiben.
}
```

2. Die Anfangswerte der einzelnen Vektorelemente können auch in einer Liste aufgezählt werden:

```
short svec[10] = { 0, 1, 17, 4, 21, 26, 7, 31, 42, 4711 };
```

Gibt man zu viele Werte an, meldet der Compiler hier einen Fehler, während er es akzeptiert, wenn man weniger Elemente angibt. Diese werden dann mit Null initialisiert.

Mehrdimensionale Vektoren werden mit entsprechend geschachtelten Elementlisten initialisiert:

```
int vvi[4][2] = { {1, 2}, {2, 3}, {3, 4}, {4, 5} };
```

Hier ist `"vvi"` (Sie erinnern sich: Deklarationen immer von innen nach außen lesen!) ein vierelementiger Vektor von Paaren von int's, also muß er auch mit vier Zahlenpaaren und nicht etwa zwei Viertupeln initialisiert werden.

Alternativ kann man solche Schachtelungen auch „plattklopfen":

```
int vvi[4][2] = { 1, 2, 2, 3, 3, 4, 4, 5 };
```

Der Effekt ist derselbe, aber natürlich ist das weniger schön.

Auch Zeichenketten kann man mit Elementlisten initialiseren, wenn man das unbedingt will:

```
char string[5] = { 'H', 'a', 'l', 'l', 'o' };
```

Bei so einer Initialisierung können Sie dann auch, wie hier geschehen, das sonst obligatorische Nulbyte am Stringende weglassen.

Ein recht angenehmes Feature von C ist (man beachte: Ich habe soeben erstmals ANSI-C gelobt!), daß man Vektorgrenzen durch die Initialisierung festlegen lassen kann. Bei der Deklaration

```
char wasndas[ ] = "Watt is'n das?";
```

wird die Stringvariable `"wasndas"` gerade so groß gewählt, daß der String einschließlich Nullbyte am Ende hineinpaßt. Analog geht das auch mit Elementlisten:

```
int Daten[ ] = { 4, 2, 42 };
```

deklariert und initialisiert einen Vektor mit drei Elementen. Sogar mehrdimensional ist das möglich:

```
int irgendwas[ ][ ][ ] = {{{1,2},{3}},{{17,4,66}},{{1,2,4}},{{99}}};
```

In der innersten Klammerebene, also auf dem dritten Indexlevel, werden hier jeweils ein, zwei oder drei Zahlen angegeben. Der Compiler nimmt dann das Maximum, also geht der dritte Indexbereich von `0` bis `2`. Auf der mittleren Ebene werden, wie Sie durch genaues Hinsehen vielleicht erkennen können, ein bis zwei Elemente initialisiert, und auf der obersten Klammerebene werden vier Initialisierungsvektoren angegeben. Nicht initialisierte Daten werden wieder mit 0 beschrieben, also kommt effektiv

```
int irgendwas[4][2][3] =  { { { {  1,  2, 0},  { 3, 0, 0} },
                              { { 17,  4, 66},  { 0, 0, 0} },
                              { { 1,   2, 4},   { 0, 0, 0} },
                              { { 99,  0, 0},   { 0, 0, 0} }
                            };
```

dabei heraus.

Wenn man bei mehrdimensionalen Vektoren die Indexbereiche wegläßt, darf man die Initialisierungen natürlich nicht wieder „plattkloppen": Bei

```
short falsch[ ][ ] = {17, 4, 21, 26, 7, 31, 42};
```

kann der Compiler ja nicht mehr herausfinden, wie groß die beiden Indexbereiche jeweils zu wählen sind. Eindeutig und deshalb erlaubt ist

```
short richtig[ ][2] = {17, 4, 21, 26, 7, 31, 42};
```

Im umgekehrten Fall,

```
short nichtrichtig[4][ ] = {17, 4, 21, 26, 7, 31, 42};
```

ist der Compiler aber nicht clever genug, die Elementanzahl durch 4 zu teilen und aufzurunden, und so ist auch hier das Ergebnis undefiniert.

Bei der hier mehrfach verwendeten Zahlenfolge handelt es sich übrigens um die Lottozahlen der nächsten Woche. Von etwaigen Gewinnen mit diesen Zahlen gehen 50 Prozent an mich.

Natürlich darf man auch Elementlisten- und Stringinitialisierung mischen:

```
char Namen[ ][ ] = { "DeGarmo", "Jackson", "Rockenfield", "Tate",
"Wilton" };
```

legt ein Feld von 5 Strings mit jeweils 12 Zeichen an, also gerade genug für unsere Daten. Das ist in der Tat ausgesprochen angenehm. Weniger angenehm ist, daß alles, was in diesem Kapitel beschrieben wurde, wirklich ausschließlich für Initialisierungen gilt und für normale Wertzuweisungen an Vektoren in C nicht erlaubt ist:

```
void main()
{
   int valsch[4];
   valsch = {0, 8, 15, 4711};      // SYNTAX ERROR
}
```

Na ja, man kann eben nicht alles haben.

2.1.3 Referenzen

2.1.3.1 Einführung

Schlafen Sie in letzter Zeit schlecht? Tauchen in Ihren Träumen ständig wüste Diagramme mit seltsamen Objekten und Zeigern darauf auf? Dann ist die Diagnose eindeutig: Sie haben die Sache mit den Pointern noch nicht so ganz verdaut. Aber Ihnen kann geholfen werden: C++ bietet Ihnen, im Gegensatz zu C, ein alternatives Datentypkonzept an, mit dem man oft Zeiger vermeiden kann - die Rede ist von Referenzen.

Intern ist ein Referenztyp nichts anderes als ein Zeigertyp, aber mit dem Unterschied, daß eine Referenz nicht wie ein Zeiger aussieht. Man deklariert Referenzen auch genau wie Zeiger, nur mit einem `"&"` statt `"*"`:

```
int &ir;
```

Damit haben wir aber auch schon den ersten Fehler gemacht: Eine Referenz muß immer initialisiert werden. Der Compiler achtet also lobenswerterweise darauf, daß eine Referenz nicht ins Irgendwo, sondern auf ein real existierendes Datenobjekt verweist. Als Initialisierung wird normalerweise ein L-Wert des referierten Typs erwartet:

```
int i = 26731, j = 3718847;
int &ir = i;
```

Im weiteren Programmverlauf zeigt `"ir"` dann unveränderbar auf die Variable `"i"` und kann deshalb synonym zu `"i"` benutzt werden:

```
ir = 42;
```

ändert über `"ir"` den Wert von `"i"`, und auch

```
ir = j;
```

setzt nicht etwa die Referenz `"ir"` auf `"j"` um, sondern ändert ebenfalls den Wert von `"i"`, so daß es bei Referenzen einen grundsätzlichen Unterschied zwischen Initialisierung und Wertzuweisung gibt. Nach der Initialisierung verweist eine Referenz also immer auf dasselbe Datenobjekt, und es gibt keine Möglichkeit, das zu ändern. Sogar wenn man die Adresse einer Referenz nimmt, etwa `"&ir"`, erhält man wieder nur die Adresse des Objekts, auf das `"ir"` zeigt, und nicht etwa die Speicherstelle, an der intern dieser Zeiger abgespeichert wird.

2.1.3.2 „VAR-Parameter" und temporäre Objekte

An dieser Stelle werden Sie natürlich fragen, was Sie denn davon haben, wenn Sie in einem Programm eine Variable über einen solchen Verweis ansprechen können. In der Form, in der Referenzen oben benutzt wurden, noch gar nichts.

Eine der nützlichsten Anwendungen für Referenztypen sind Funktionsparameter. Vielleicht erinnern Sie sich noch, daß bei der Argumentauswertung dieselben Regeln wie bei einer Initialisierung gelten. Wenn Sie also einen Parameter als Referenz auf einen Typen T deklarieren, wird beim Funktionsaufruf als Argument ein L-Wert des Typs T erwartet und die Referenz automatisch mit einem Verweis auf diese Variable initialisiert:

```
void halbiere (int &para)
{ para /= 2;
}

void main()
{ int i = 84;
  halbiere (i);
}
```

Die Funktion `"halbiere"` hat also einen Seiteneffekt auf die als Argument übergebene Variable. Pascal-Programmierer kennen dieses Feature als „VAR-Parameter", nur daß man in C++ Referenzen eben universell benutzen kann und nicht nur in Parameterlisten.

In diesem Zusammenhang gibt es ein ziemlich eigentümliches Feature, das im 1.0-Standard eher versehentlich enthalten war und vom 2.0-Standard nur noch „inoffiziell" unterstützt wird (d. h. es gehört offiziell nicht mehr dazu, aber es wird empfohlen, es trotzdem noch zu unterstützen, um die Kompatibilität der Programme zu erhalten): Der Ausdruck, mit dem eine Referenz initialisiert wird, muß weder ein L-Wert sein noch genau den richtigen Typ haben.

Beispielsweise könnte man die Funktion `"halbiere"` auch folgendermaßen aufrufen:

```
halbiere(42);
```

Was soll der Quatsch? Wird jetzt die Konstante `"42"` halbiert (was bekanntlich alle Probleme des Universums halbieren würde)? Nein, wir schließen hier Bekanntschaft mit einem Feature namens „temporäres Objekt". Der Compiler richtet hier ganz ohne Zutun des Programmierers eine namen-

lose "int"-Variable ein, initialisiert diese mit 42 und ruft "halbiere" dann auf dieser Phantom-Variablen auf. Das ist zwar noch ein recht nettes Feature, aber es wird ziemlich eklig, wenn beim Aufruf eine Typkonvertierung nötig wird:

```
long x = 12;
halbiere(x);
```

"x" ist zwar ein L-Wert, aber eben nicht vom Typ "int", und deshalb ist hier eine Typkonvertierung nötig, und das Ergebnis einer solchen Operation ist generell kein L-Wert, und zwar nicht einmal unter MaxonC++, wo die interne Darstellungen von "int" und "long" identisch sind (jeweils 32 Bit) und deshalb eigentlich keine wirkliche Operation ausgeführt werden muß. Jedenfalls generiert MaxonC++ auch hier ein temporäres Objekt, initialisiert es mit "x" und ruft die Funktion "halbiere" auf, die das temporäre Objekt halbiert - und der Programmierer wundert sich höchstwahrscheinlich, daß der Wert von "x" nicht halbiert wird (was im obigen Beispiel totsicher gemeint war).

Zu allem Überfluß funktioniert dieses Feature nicht nur bei Funktionsparametern, sondern bei sämtlichen Referenzen, z. B.

```
int &ir = 3.14159;       // Etwas komisch, aber erlaubt
char c;
signed char &cr = c;     // Überraschung!!!
```

Das Beispiel mit dem "ir" ist zwar ziemlich obskur. Hier wandelt der Compiler 3.14159 nach "int" um, initialisiert ein temporäres Objekt damit und läßt "ir" darauf verweisen, was natürlich ausgesprochen unnütz ist. Die Initialisierung von "cr" hingegen steckt voller Heimtücke. Da "char" und "signed char" in C++ zwei unterschiedliche Typen sind (obwohl auch sie in MaxonC++ praktisch identisch sind), verweist "cr" hier nicht etwa auf "c", sondern auf ein temporäres Objekt.

Im 2.0-Standard ist die Angelegenheit wesentlich eleganter gelöst. Ich muß an dieser Stelle etwas vorgreifen (nämlich auf Abschnitt 2.2.3) und schon jetzt verraten, daß man Typen aller Art mit "const" qualifizieren kann, etwa so:

```
void machwas(const double &D);
```

Die Referenz verweist hier auf ein "const double", was nichts anderes bedeutet, als daß man nicht beabsichtigt, den Wert jenes Datenobjekts zu verändern, wobei der Compiler das auch überwacht und jegliche Veränderung des referierten Objekts als Fehler meldet. Nun ist es ziemlich unerheblich, ob die Referenz ein direkter Zeiger auf das Funktionsargument oder nur ein Verweis auf ein temporäres Objekt ist, denn hier sind Überraschungen weitgehend ausgeschlossen: Da die Funktion das referierte Objekt nicht verändern kann, hat sie auch keine Seiteneffekte darauf, und der Programmierer wird nicht mehr mit der Tatsache konfrontiert, daß anstelle des Arguments unter Umständen bloß ein temporäres Objekt verändert wird.

Jedenfalls lautet die neue, korrekte Sprachregel so, daß eine Referenz auf ein nicht-konstantes Objekt mit einem L-Wert des richtigen Typs initialisiert werden muß, während bei Referenzen auf mit „const" qualifizierte Typen (und nur hier!) bei Bedarf temporäre Objekte eingerichtet werden.

2.1.3.3 Weitere Möglichkeiten und Einschränkungen

Zeiger auf Referenzen sind ebenso wenig erlaubt wie Vektoren von Referenzen:

```
int &*p1;        // ERROR, Zeiger auf Referenz
int &v1[10];     // ERROR, Vektor von Referenzen
```

Umgekehrt geht das natürlich:

```
int *&p2;        // Referenz auf Zeiger
int (&v2)[10];// Referenz auf Vektor
```

Eine Funktion darf auch eine Referenz zurückgeben:

```
int &f(int i)
{ int j = 2*i;
  return j;
}
```

Diese Funktion ist zwar formal richtig, aber trotzdem falsch: Hier wird ein Verweis auf die lokale Variable `"j"` zurückgegeben, die nach Ausführung der Funktion nicht mehr existiert. Also zeigt die Referenz irgendwo in die Pampa, und eine Zuweisung an das Ergebnis der Funktion, etwa

```
int &f(int i);// Liefert wie oben Referenz auf ein "int"

void main()
{ f(1) = 42;  // Ändert Wert des Objekts, auf das "f(1)" verweist
}
```

hätte unter Umständen fatale Folgen. So ganz sind Sie also auch bei Referenzen nicht gegen Pointer-Chaos gefeit.

Eine etwas sinnvollere Anwendung für Funktionen mit Referenz-Ergebnis wäre z. B. ein simulierter Vektor. Manchmal hat man eine Liste von Daten nicht als richtigen Vektor vorliegen, möchte sie aber trotzdem wie Vektorelemente benutzen. Das könnte so aussehen:

```
int i0, i1, i2, i3;   // Unsere vier Daten

int &Liste (int Index)
{ switch (Index)
  { case 0:
      return i0;
    case 1:
      return i1;
    case 2:
      return i2;
    case 3:
      return i3;
```

```
    default:    // ... Fehlermeldung ausgeben, Programm abbrechen ...
  }
}

void main()
{ int i;
  for (i=0; i<4; i++)
    Liste(i) = 42+i;

  Liste(0) = 2*Liste(1)+Liste(3);
}
```

Wie Sie sehen, können wir auf diese Weise unsere ganz normalen Variablen `"i0"` bis `"i3"` wie einen Vektor benutzen. Sinnvoll ist eine solche Konstruktion aber vor allem dann, wenn die fraglichen Daten nicht in Form von gewöhnlichen Variablen, sondern z. B. als dynamisch erzeugte verkettete Liste vorliegen - aber das gehört in ein anderes Kapitel. Vorher müssen wir noch unsere Hausaufgaben machen und uns durch diverse Features kämpfen, die uns C++ sonst noch so bietet.

2.2 Qualifizierungen, Deklarationen und Spezifikationen

2.2.1 Speicherklassen

2.2.1.1 Automatische und statische Daten

Ein C-Programm hat verschiedene Möglichkeiten, Daten abzuspreichern. Da wäre beispielsweise der Stack, ein RAM-Bereich, der für jeden Prozess eingerichtet wird und dann, wie der Name schon sagt, wie ein Stapel funktioniert: Man kann Daten drauflegen und sie später wieder herunternehmen.

In C-Programmen wird der Stack vor allem für lokale Daten von Funktionen und für ihre Argumente benutzt. Wenn eine Funktion benutzt wird, legt der aufrufende Programmteil die Argumente auf den Stack und ruft die Funktion als Unterprogramm auf, wobei der Prozessor dann übrigens auch noch die Rückkehradresse, an der er nach Beendigung der Funktion weitermachen soll, ebenfalls auf dem Stack vermerkt. Die Funktion richtet sich dann selbst genug Stackplatz für ihre lokalen Daten ein. Am Ende der Funktion, etwa bei einem `"return"`-Statement, werden die lokalen Daten wieder vom Stack entfernt und dann die Rückkehradresse heruntergenommen und angesprungen, so daß der Prozessor wieder im aufrufenden Programm landet. Diesem obliegt es dann, die Argumente wieder vom Stack zu entfernen.

Da C eine Hochsprache (oder zumindest als solche gedacht) ist, müssen Sie sich um solche Mechanismen natürlich nicht selbst kümmern. Wichtig ist nur, daß Ihnen die prinzipiellen Mechanismen, mit denen solche Variablen erzeugt und wieder zerstört werden, klar sind. In den Abschnitten 2.1.1.3 und 2.1.3 wurde ja bereits gesagt, was passieren kann, wenn man diese Interna mißachtet und Zeiger bzw. Referenzen auf lokale Daten aus einer Funktion exportiert.

Weil die Variablen auf dem Stack aus Sicht des C-Programmierers automatisch erzeugt und gelöscht werden, spricht man in diesem Zusammenhang auch von „automatischen" Variablen. Das Gegenteil davon sind „statische" Datenobjekte.

Statische Daten werden natürlich auch vollautomatisch eingerichtet, aber das nur einmal, nämlich am Programmanfang. Sie existieren dann bis zum Programmende. Globale Variablen, also solche, die außerhalb von Funktionen deklariert werden, sind grundsätzlich statisch.

Bei jeder Variablendeklaration (und bei einigen anderen Deklarationen auch) ist es möglich, die Speicherklasse explizit anzugeben. Dazu dienen hauptsächlich die Schlüsselwörter `"auto"` und `"static"`:

```
int f()
{ auto int i, j;
  static int k;
}
```

Die beiden Variablen `"i"` und `"j"` werden jedesmal, wenn die Funktion `"f"` ausgeführt wird, neu eingerichtet und bei Beendigung der Funktion wieder vernichtet. Das Wortsymbol `"auto"` ist übrigens generell nutzlos und könnte ebenso gut ersatzlos gestrichen werden, und das macht man in der Praxis dann auch. Da keine Initialisierung angegeben ist, haben `"i"` und `"j"` zunächst undefinierte Werte.

Dadurch, daß der Deklaration von `"k"` das Schlüsselwort `"static"` vorangestellt wird, verhält diese Variable sich genau so, als wäre sie global, also auf Dateiebene, deklariert worden, mit einem wichtigen Unterschied: Sie ist auf diese Art und Weise ausschließlich innerhalb der Funktion `"f"` bekannt und benutzbar.

Alle statischen Daten werden beim Programmstart initialisiert, und zwar mit Null, sofern der Programmierer keine andere Initialisierung angibt. Wenn Sie beispielsweise am Anfang einer Funktion

```
static int count;
++count;
```

deklarieren, kann sie immer anhand von `"count"` feststellen, zum wievielten Mal sie aufgerufen wurde.

Das Symbol `"auto"` darf ausschließlich innerhalb von Funktionen verwendet werden, wo es bekanntlich überflüssig ist, da lokale Variablen ja sowieso schon „automatisch" sind, sofern nicht eine andere Speicherklasse spezifiziert wird. `"static"` kann im hier beschriebenen Sinne ebenfalls nur innerhalb von Funktionen benutzt werden. Auf Dateiebene, wo Variablen ja defaultmäßig statisch sind, hat es verwirrenderweise eine etwas andere Bedeutung, die im folgenden Abschnitt beschrieben wird.

ANSI C verlangt, daß statische Daten ausschließlich mit konstanten Werten initialisiert werden. C++ ist da weniger pingelig und läßt auch beliebige nicht-konstante Ausdrücke zu, die auch andere statische Variablen und sogar Funktionsaufrufen enthalten dürfen, beispielsweise

```
double f(int n);

int i = 0;
double d = f(i);      // C++, aber nicht C
long l = (long) d;    // Ebenfalls nur in C++
```

Undefinierte Ergebnisse erhält man dagegen, wenn man statische Variablen mit nicht-statischen Werten initialisiert, etwa

```
void f(int i)
{ static int j = i;  // Falsch!
}
```

Wie bereits gesagt, geschieht die Initialisierung statischer Daten am Programmbeginn, also zu einem Zeitpunkt, an der hier die Variable `"i"` noch nicht initialisiert ist, so daß `"j"` hier mit einem undefinierten Wert initialisiert wird.

2.2.1.2 Module und ihre Gültigkeitsbereiche

Gemeinsame Daten und Funktionen

Bisher habe ich Ihnen diskret verschwiegen, daß ein C-Programm ohne weiteres aus mehreren Modulen bestehen kann. Jedes Modul entsteht aus einem Quelltext (eventuell einschließlich Includedateien) und kann unabhängig von den anderen Modulen compiliert werden, weshalb man ein Modul in C etwas verschämt auch „Übersetzungseinheit" („translation unit") nennt. „Verschämt" deshalb, weil das Modulkonzept nicht besonders schön in die Programmiersprache C integriert wurde. Andere Hochsprachen, etwa Modula oder die allseits bekannten modernen Pascal-Dialekte, ermöglichen nämlich durch geeignete Sprachkonstrukte eine klare Trennung zwischen dem „Interface" eines Moduls, also dem Teil, der für andere Module sichtbar sein soll, und der „Implementierung", also den Programmteilen, die Privatsache des Moduls sein sollen. In C hat man ganz einfach festgelegt, daß jede Funktion und jedes auf Dateiebene deklarierte Datenobjekt öffentlich sind und von anderen Modulen benutzt werden können.

Besonders einfach ist das bei Funktionen: Wird eine Funktion deklariert, aber nicht definiert, weiß der Compiler, daß er sie vor irgendwo importieren muß. Beispielsweise könnte ein Modul folgendes enthalten:

```
// Datei "hello.c"

#include <stream.h>

void hello()
{ cout « "Hello, World!\n";
}
```

Dann braucht ein anderes Modul die Funktion nur noch zu deklarieren und kann sie dann benutzen:

```
// Modul "haupt"

void hello();

void main()
{ hello();
}
```

Natürlich muß man dem Linker irgendwie klarmachen, daß er das Programm aus den beiden Modulen `"hello"` und `"haupt"` zusammensetzen soll. In der Entwicklungsumgebung von MaxonC++ muß man dazu ein „*Makefile*" erzeugen, bei der Kommandozeilenversion reicht es, wenn man beide Quelldateinamen gleichzeitig als Parameter angibt. Näheres dazu steht im ersten Teil dieses Handbuchs.

Der Linker verlangt dann jedenfalls, daß irgend ein Modul eine Funktion `"hello"` exportiert. Es ist natürlich möglich, daß sie in mehreren Übersetzungseinheiten zugleich definiert wird. Dann meldet der Linker einen entsprechenden Fehler, weil er dann ja nicht weiß, welches `"hello"` er nun nehmen soll.

Bei gemeinsam benutzten Daten ist das Ganze etwas komplizierter, denn eine Deklaration wie `"int i;"` teilt dem Compiler bekanntlich nicht bloß mit, daß es eine Variable `"i"` gibt, sondern weist ihn auch an, eine solche zu erzeugen. Strenggenommen handelt es sich dabei also nicht um eine Deklaration, sondern eine Definition, aber der Begriff „Variablendeklaration" wird nun einmal traditionell benutzt. Jedenfalls hat jedes Modul, in dem `"int i;"` deklariert wird, sein eigenes `"i"`, daß es zu allem Überfluß auch noch exportiert, so daß der Linker sich beklagen wird, daß dieses Symbol mehrfach definiert wird. So geht es also nicht.

Die Lösung dieses Problems ist die Speicherklasse `"extern"`. Dadurch wird der Compiler angewiesen, ein Datenobjekt nicht zu erzeugen, sondern es zu importieren. `"extern"` wird syntaktisch genau wie `"static"` oder `"auto"` benutzt:

```
extern int i;
```

Man darf innerhalb einer Übersetzungseinheit jede Variable beliebig oft extern und zusätzlich höchstens einmal „normal" deklarieren, z. B.

```
extern int i;
int j;
extern int i, j;
int i;
```

Das erleichtert es enorm, die Schnittstelle zwischen Modulen in einem „Header-File" zu deklarieren. Normalerweise legt man zu jeder Übersetzungseinheit eine Includedatei an, in der alles deklariert wird, was vom jeweiligen Modul der „Allgemeinheit" zur Verfügung gestellt wird. Nehmen wir an, das Modul „*hello.c*" exportiere die Funktion `"hello"` und die Strungvariable `"text"`. Das „Headerfile" dazu, das gewöhnlich „*hello.h*" oder „*hello.i*" genannt wird, enthält dann die Prototypen der exportierten Funktionen und `"extern"`-Deklarationen für die Variablen:

```
// * Header"hello.h" *
extern char text[];
void hello();
```

Das Modul `"hello"` definiert dann diese Namen. In die Schnittstellendeklaration gehören normalerweise nicht nur Variablen- und Funktionsdeklarationen, sondern auch Typdefintionen, die Gegenstand einiger folgender Kapitel sind. Deshalb muß im allgemeinen das Modul sein eigenes

„Headerfile" einschließen, wobei der Dateiname allerdings in doppelten Hochkomma `"..."` und nicht wie bisher in spitzen Klammern `<...>` zu setzen ist, denn es handelt sich diesmal ja nicht um eine Standarddatei:

```
// * Modul "hello.c" *

#include "hello.h"    // Eigenes Headerfile
#include <stream.h>   // Wie gewohnt

char text[] = "Hello, World!";

void hello()
{ cout « text;
}
```

Das Hauptprogramm, nennen wir es *„main.c"*, wird in der Regel nichts exportieren und braucht deshalb kein eigenes „Headerfile". Es benutzt lediglich das von `"hello"` und hat dann alle Deklarationen, die es braucht, um die Funktionen von `"hello"` zu benutzen:

```
// * Hauptmodul *

#include "hello.h"

void main()
{ char *txtptr = text;
  hello();
}
```

Übrigens darf man das Schlüsselwort `"extern"` auch vor Funktionsdeklarationen setzen, z. B.

```
extern void hello();
```

Das ist aber überflüssig, denn Funktionen, die nicht definiert werden, gelten ja automatisch als `"extern"`.

Modulinterne „static"-Namen

Objekte und Funktionen, die auf Dateiebene deklariert werden, exportieren in der Regel ein Namenssymbol an den Linker, so daß alle anderen Module darauf zugreifen können. Manchmal führt das zu unerwarteten Effekten, wenn zwei Übersetzungseinheiten denselben Bezeichner für unterschiedliche modulinterne Zwecke definieren. Der Compiler stört sich daran nicht (denn er betrachtet ja immer nur eine Übersetzungseinheit, also ein Modul), aber der Linker mäkelt, weil ein Name mehrfach definiert wurde. Um derartige Kollisionen von vornherein zu verhindern, sieht C die Möglichkeit vor, Variablen und Funktionen als „nicht-exportiert" zu deklarieren. Aus nicht mehr nachvollziehbaren Gründen hat man sich entschieden, dafür das Schlüsselwort `"static"` zu überladen: Eine Deklaration wie

```
static char c;
```

hat schon in ANSI-C zwei Bedeutungen (und in C++ noch eine dritte, dazu an anderer Stelle mehr), je nach dem, wo sie steht:

- In einer Funktion deklariert sie eine Variable, die nicht auf dem Stack (im „automatischen" Speicher), sondern statisch im Datenbereich des Programms angelegt wird und dort die ganze Programmlaufzeit über existiert.
- Auf Dateiebene, also außerhalb aller Funktionsdefinitionen, deklariert sie ebenfalls eine Variable im statischen Speicher (denn alle Variablen auf Dateiebene sind statisch), deklariert aber zusätzlich, daß dieses Datenobjekt nicht anderen Modulen zugänglich sein soll. Es wird dafür insbesondere kein Symbol an den Linker weitergereicht, so daß es auch nicht zu versehentlichen Kollisionen kommen kann.

Bitte fragen Sie mich nicht, wieso man `"static"` für zwei ganz unterschiedliche Zwecke benutzt - ich weiß es nicht, und wenn ich etwas zu sagen hätte, würde man für letztere Funktion ein neues Schlüsselwort wie `"intern"`, analog zu `"extern"`, einführen, aber mich fragt ja keiner.

Jedenfalls ist man insofern konsistent, als daß man auch Funktionen als `"static"` deklarieren kann, etwa so:

```
static void hello();
```

Als kleine Erleichterung muß man das `"static"` nur bei der ersten Deklaration der Funktion hinschreiben, bei allen folgenden darf man es auch weglassen:

```
static void hello();

void hello()      // OK
{ // usw..
}
```

Umgekehrt ist es allerdings verboten:

```
void hello();

// ... irgendwas ...

static void hello(); // Error: "Inconsistent linkage"
```

Das liegt daran, daß der Compiler sich bei der ersten Deklaration schon psychisch und moralisch darauf eingestellt hat, die Funktion notfalls einlinken zu dürfen, und erst später mitgeteilt bekommt, daß er eben das doch nicht darf. Immerhin könnte ja da, wo jetzt `"irgendwas"` steht, bereits ein Funktionsaufruf einschließlich Symbolimport stattgefunden haben, und C und C++ sind so konzipiert, daß der Compiler das Programm in einem Pass übersetzen kann, ohne noch einmal zurückzugehen.

Typsicheres Linken und Link-Spezifikationen

Es war bereits mehrfach davon die Rede, daß „Symbole" an den Linker übergeben werden. Das sieht dann so aus, daß der Name des Objekts bzw. der Funktion dem Linker bekannt gemacht wird, ver-

sehen mit einem Vermerk, ob die Objekt-Datei das Symbol definiert (zusammen mit dem Symbolwert, also der Speicheradresse) oder benutzt (und einer Tabelle mit den Stellen, an denen dieses Symbol referiert wird). Allerdings wird der Name nicht ganz so exportiert, wie er definiert wird. Bei Variablen wird dem Bezeichner lediglich ein Unterstrich `"_"` vorangestellt, d. h. eine Variable `"Daten"` heißt für den Linker `"_Daten"`. Dies ist eine allgemeine Konvention bei C-Compilern.

Es gibt übrigens im ANSI C-Standard eine mittlerweile ziemlich antiquierte Definition, nach der ein Linker nur die ersten sechs Zeichen eines Namens beachten und dabei auch nicht zwischen Groß- und Kleinbuchstaben unterscheiden muß. Demnach kann es theoretisch Systeme geben, auf denen die Variablen `"String"` und `"stringlaenge"` für den Linker identisch sind. Mir ist aber kein solches System bekannt, und MaxonC++ beachtet wie jedes heute gebräuchliche C-System den ganzen Namen unter Unterscheidung von Groß- und Kleinbuchstaben.

In C++ ist es bekanntlich möglich, Funktionen zu überladen. Deshalb reicht es nicht, wenn man dem Linker lediglich den Namen einer Funktion übergibt, denn er muß ja in der Lage sein, die beiden Funktionen

```
void fun(int);
```

und

```
char *fun(char*);
```

zu unterscheiden. Dieses Problem wird gelöst, indem die Parameterliste in das Linkersymbol hineinkodiert wird:

Auf den Funktionsbezeichner (diesmal ohne vorangestellten `"_"`) folgen ein Unterstrich und eine Kodierung der Parametertypen. Dabei sind jedem Grundtyp ein oder zwei Buchstaben zugeordnet:

```
Typ  Code


char c
signed char Sc
unsigned char Uc
short int s
unsigned shortUs
int  i
unsigned int  Ui
long int  j
unsigned long Uj
long long l
unsigned long long Ul
float  f
double d
long double D
void v
```

Der Ergebnistyp der Funktion wird nicht kodiert, denn C++ erlaubt es nicht, zwei Funktionen mit identischen Parameterlisten, aber unterschiedlichem Rückgabetyp zu deklarieren. Ein Zeigertyp

wird kodiert, indem dem Code des referierten Typs ein `"P"` vorangestellt wird. Aus `"char*"` wird also `"Pc"` und `"PPv"` ist ein Zeiger auf einen Zeiger auf `"void"`, also `"void**"`. Entsprechend stehen `"R"` für Referenzen und `"a"` (mit zusätzlich kodierter Elementanzahl) für Vektoren. Die folgende Übersicht enthält auch schon einmal Codes für Datentypen, die in diesem Handbuch erst später eingeführt werden:

`Pointer`	**P**
`Referenz`	**R**
`Vektor`	**A + Elementanzahl**
`const`	**C**
`volatile`	**V**
`Funktion`	**F + Parameterliste + f**

Einige Datentypen können nur über ihren Namen kodiert werden:

`Class, Struct, Union`	Namenslänge (2 Ziffern) + Name
`Enumeration`	E + Namenslänge (2 Ziffern) + Name

Die Funktion `"strcmp(char*, char*)"` heißt dann z. B. für den Linker `"strcmp_PcPc"`. Enthält der Funktionsname selbst einen Unterstrich, etwa `"Input_data"`, wird dieser im Linkersymbol verdoppelt: `"Input__data"`. Dadurch kann es nicht zu Kollisionen zwischen Unterstrichen im Funktionsnamen und dem Unterstrich zwischen Namen und Parameterliste kommen. Eine Ellipse `"..."` am Ende der Parameterliste wird durch ein angehängtes `"e"` dargestellt.

Das Kodieren der Parameterliste ist aber nicht nur eine Notlösung, um überladene Funktionen mit dem Linker behandeln zu können, sondern hat auch den angenehmen Effekt, daß auch über Modulgrenzen hinweg sichergestellt ist, daß Funktionsargumente den richtigen Typ haben. Deshalb nennt man diesen Mechanismus auch „typsicheres Linken".

ANSI C kennt kein Überladen, deshalb werden hier Funktionsnamen in der Regel genau wie Variablennamen kodiert. Auch MaxonC++ hält sich im C-Modus an diese Konvention. Dadurch wird es natürlich schwierig, C- und C++-Module zu verbinden, weshalb in C++ das Konzept der Link-Spezifikationen eingeführt wurde.

Bei einer Linkspezifikation folgt dem Schlüsselwort `"extern"` die gewünschte Programmiersprache als String, z. B.

```
extern "C" void test(int);
```

Diese Deklaration weist den Linker an, die Funktion `"test"` beim Linken nach C-Regeln zu behandeln, also `"_test"` statt `"test_i"`. Jeder C++-Compiler kennt die Linkmodi „C" und „C++", in MaxonC++ gibt es auch noch „Asm" und „Pascal", wo dann das Symbol mit dem Bezeichner identisch ist und weder ein `"_"` am Anfang noch eine kodierte Parameterliste bekommt. Da es aber langweilig ist, vor zehntausend Deklarationen jedesmal diese Spezifikation zu schreiben, kann man Deklarationen auch klammern:

```
extern "Asm"
{ void test(int);          // Externes Symbol: "test",
  int Anzahl, Maximum;  // "Anzahl" und "Maximum",
  char* input();           // "input"
}
```

Wem es Spaß macht, der darf Linkspezifikationen auch ineinander verschachteln. Die Linkspezifikation müssen jeweils die erste Deklaration einer Funktion sein, sonst weiß der Compiler wieder nicht, was er soll. Bei weiteren Deklarationen bzw. Definitionen der Funktion muß man dann den Linkmodus nicht noch einmal angeben.

Die Schnittstelle zum Amiga-Betriebssystem

MaxonC++ kennt außer „C", „C++", „Asm" und „Pascal" noch den Linkmodus „Amiga", der eigentlich keiner ist. Dazu muß man erst einmal wissen, wie das Amiga-Betriebssystem aufgebaut ist.

Das Amiga OS besteht aus einer Sammlung von Libraries, also Bibliotheken oder Funktionssammlungen. Dabei vollzieht sich der Einsprung in eine solche Funktion etwas eigenartig. Zunächst einmal hat jede Library eine Basis-Struktur, die gewisse globale Daten der Bibliothek enthält. Wenn man eine Bibliotheksfunktion aufruft, wird verlangt, daß im Prozessorregister `"a6"` ein Zeiger auf eben diese Struktur steht. Vor der Basis-Struktur steht im Speicher eine Sprungtabelle, in der Sprunganweisungen zu den jeweiligen Bibliotheksfunktionen stehen.

Um eine Systemfunktion aufzurufen, lädt man einen Zeiger auf die Basis-Struktur nach `"a6"` und springt dann an einen gewissen (negativen) Offset unterhalb der Struktur. Die Argumente sind dabei in der Regel nicht auf dem Stack, sondern in Prozessorregistern zu übergeben, aber dazu mehr im folgenden Abschnitt. Jedenfalls werden Betriebssystemfunktionen in MaxonC++ nicht vom Linker eingebunden, sondern direkt angesprungen. Dazu benötigt der Compiler allerdings zwei Angaben: Die Basisadresse und den Offset der Funktion. MaxonC++ bietet dazu (auch im C-Modus) eine besondere Deklarationsweise, die an die Link-Spezifikationen angelehnt ist. Hinter dem Wortsymbol `"extern"` und dem Linkmodus „Amiga" folgt dabei zunächst der Name einer Variablen, die die Basisadresse der Library enthalten soll. Hinter der eigentlichen Funktionsdeklaration ist dann der jeweilige Offset mit einem `"="` anzugeben.

Bei der „Exec"-Library sieht das zum Beispiel so aus:

```
extern "AmigaLib" SysBase
{ void InitCode (register unsigned d0, register unsigned d1) =-0x48;
  void InitStruct (register APTR a1,register APTR a2,
                   register unsigned d0) =-0x4e;
  struct Library *MakeLibrary (  register APTR a0, register APTR a1,
                                 register unsigned (*a2)(),
                                 register unsigned d0,
                                 register unsigned d1) =-0x54;
  void MakeFunctions (  register APTR a0, register APTR a1,
                        register unsigned a2) =-0x5a;
  struct Resident *FindResident (register UBYTE *a1) =-0x60;
  void InitResident (   register struct Resident *a1, register
                        unsigned d1) =-0x66;
```

```
  void Alert (register unsigned d7) =-0x6c;
  void Debug (register unsigned d0) =-0x72;
  void Disable (void) =-0x78;
  void Enable (void) =-0x7e;
  void Forbid (void) =-0x84;
  void Permit (void) =-0x8a;   // usw.
}
```

Wahrscheinlich werden Sie solche Deklarationen aber niemals selbst schreiben müssen, denn sie sind ja für alle Amiga-Libraries bereits vorhanden. Alternativ akzeptiert MaxonC++ auch die `"#pragma amicall"`-Schreibweise, wie sie von anderen C-Compilern bekannt ist. Diese wird im „Preprozessor"-Kapitel dargestellt.

Möglicherweise haben Sie sich oben über die seltsame Parametersyntax mit dem `"register"` gewundert. Das bietet dann auch schon die ideale Überleitung zum nächsten Abschnitt:

2.2.1.3 Daten in Registern

Jeder Prozessor verwaltet nicht nur das RAM, sondern hat darüber hinaus auch einige interne Prozessorregister, in dem in der Regel Zwischenergebnisse von Berechnungen o. Ä. abgelegt werden. Der Zugriff auf diese Register geht ganz erheblich schneller als eine RAM-Operation, und da die Prozessoren der 68000er-Familie mit Registern vergleichsweise üppig ausgestattet sind (ganz im Gegensatz zu diesen INTEL-Dingern), liegt der Wunsch nahe, häufig benutzte Variablen in den nicht anderweitig benutzten Registern abzulegen und so eine ordentlichen Geschwindigkeitssteigerung zu erzielen.

Das klingt zwar alles nach maschinennaher Bitpopelei, aber erfreulicherweise hat man im ANSI C-Standard eine flexible und maschinenunabhängige Schreibweise dafür eingeführt. Den Schlüssel dazu bietet eine Speicherklasse namens `"register"`, die analog zu `"extern"`, `"auto"` oder `"static"` benutzt wird.

Register-Variablen dürfen nur innerhalb von Funktionen deklariert werden. Ansonsten ist man in ihrer Verwendung weitgehend frei: Man darf beliebig viele Variablen beliebiger Typen als `"register"` deklarieren, wobei der Compiler dann entscheidet, was er wirklich in einem (und falls ja, in welchem) Register ablegt. MaxonC++ kann ganzzahlige und Pointer- bzw. Referenzvariablen in Registern unterbringen, wobei die numerischen Typen in Daten-, Zeigertypen in Adressregistern abgespeichert werden.

Auch hier gibt es wieder einen kleinen Unterschied zwischen C und C++: In ANSI C darf man nie die Adresse einer Variablen, die als `"register"` deklariert wurde, nehmen, also den unären Adressoperator `"&"` nicht anwenden. C++ ist da großzügiger: Wird die Adresse einer Registervariablen genommen, wird diese eben doch nicht in einem Prozessorregister, sondern ganz normal auf dem Stack abgelegt.

Registerdeklarationen bringen manchmal in der Tat einiges, die folgende Funktion läuft durch das `"register"` z. B. glatt doppelt so schnell:

```
void test()
{ register int i, j=0;

  for (i=0; i<500000; ++i)
    ++j;
}
```

Man sollte sich aber hüten, allzu ausgiebig von Registern Gebrauch zu machen, denn für praktisch alle Operationen braucht der Prozessor mindestens ein Register, und wenn zu viele davon für Variablen verwendet werden, muß er diese manchmal kurzfristig auf den Stack auslagern und nachher wieder laden, so daß man natürlich keinen Geschwindigkeits-

zuwachs erwarten kann. In jeder Funktion sollte man maximal vier ganzzahlige und zwei Pointervariablen als `"register"` deklarieren. Im Zweifelsfall sollte man in MaxonC++ möglichst ganz auf Register-Variablen verzichten und die Optimierungen einschalten, denn dann entscheidet der Compiler selbstständig, welche Variablen er in Register lädt.

Als kleine Erweiterung des C++-Standards bietet MaxonC++ die Möglichkeit, auch Parameter in Registern zu übergeben. Auch dies kann bisweilen zu gewissen Geschwindigkeitssteigerungen führen. Bei der Funktionsdeklaration ist dazu einfach das Symbol `"register"` vor die Parameter zu setzen, z. B.

```
int Add (register int i, register int j);
```

Dies entspricht aber nicht dem C++-Standard! Man sollte dieses Feature also möglichst vorsichtig einsetzen, da die Programme sonst nicht mehr portabel sind. Am besten benutzt man solche Parameter nur, wenn man fertige Funktionen, die ihre Argumente in Registern erwarten, irgendwie in ein Programm einbinden will, z. B. die Amiga-Betriebssystemfunktionen oder Assembler-Routinen, die sich ja ohnehin schlecht portieren lassen.

Es gibt sogar zwei Möglichkeiten, dem Compiler vorzuschreiben, in welche Register er die Parameter legen soll: Entweder bennent man die Parameter-Variable nach einem 68000er-Register, z.B.

```
void f(register int d0, register char *a2) ...
```

oder man setzt den gewünschten Registername mit `"_"` hinter das Schlüsselwort `"register:"`

```
void f(register _d0 int i, register _a2 char s) ...
```

Wenn man aber auf diese Weise dem Compiler schon Vorschriften macht, welche Register er für Variablen verwenden soll, sollte man sich vorher genau überlegen, was man da tut.

Für selbstgeschriebene C++ Funktionen ist dieses Feature so gut wie tabu, z. B. würde die oben deklarierte Funktion unter Umständen sehr langsam, weil sie möglicherweise das Register `"d0"` ständig braucht und den Inhalt der Register-Variablen dann jedesmal retten und anschließend wiederherstellen muß. Man sollte solche Register-Festlegungen ausschließlich zur Deklaration importierter Assembler- und Systemfunktionen benutzen.

2.2.1.4 Daten im CHIP-RAM

Die Architektur des Amiga ist etwas unkonventionell: Es gibt zwei, manchmal sogar drei verscheidene Klassen von RAM. Einerseits gibt es das Chip-RAM, auf das nicht nur der Prozessor, sondern auch die diversen DMA-Kanäle zugreifen können. Deshalb müssen hier alle Grafik- und Sounddaten stehen. Der übrige Speicher wird Fast-RAM genannt, weil er ausschließlich dem Prozessor zur Verfügung steht, weshalb Zugriffe auf diesen Speicher theoretisch schneller sein könnten. Auf normalen Amigas ist das aber nicht der Fall, so daß die Bezeichnung „Fast-RAM" eigentlich gelogen ist. Auf Amigas mit 32-Bit-Prozessoren (68020, 68030, 68040, ...) gibt es aber meist außerdem noch „echtes" Fast-RAM, nämlich unabhängig vom normalen Systembus getakteten, 32 Bit breiten Speicher. Deshalb muß man genaugenommen zwischem „echtem" und „falschem" Fast-RAM unterscheiden.

In der Praxis interessiert den Programmierer aber nur die Frage „Chip oder nicht Chip", denn er muß dafür sorgen, daß Grafikdaten im Chip-Memory abgelegt werden. Unter MaxonC^{++} ist das einfach: Es gibt dafür die Pragmas `"chip"` und `"fast"`.

Alle nach der Zeile

```
#pragma chip
```

deklarierten statischen Daten werden ins Chip-Ram gepackt,

```
#pragma fast
```

schaltet wieder in den normalen Modus, in dem Daten irgendwo abgelegt werden, bevorzugt aber im Fast-Memory. Ein typsiches Beispiel ist eine Image-Struktur unter Intuition: Die Bitmap des Images dabei immer im Chip-Memory liegen.

```
#pragma chip

unsigned int BMap[] =      // Bilddaten ins CHIP-RAM
{  0x3FFC,
   0x7FFE,
   0xF00F,
   0xF00F,
   0xF00F,
   0xF00F,
   0x7FFE,
   0x3FFC
};

#pragma fast

// Die Image-Struktur selbst darf überall liegen:
struct Image MyImage = { 0, 0, 16, 8, 1, BMap, 1, 1, 0};
```

Auf automatische Daten hat `"pragma chip"` natürlich keinen Einfluß, denn diese werden generell auf dem Stack eingerichtet, und der liegt im allgemeinen nicht im Chip-Memory.

2.2.2 Typdefinitionen

Sicher haben Sie schon bemerkt, daß die Typschreibweisen in C einem ganz schön auf die Nerven gehen können, man denke nur an Dinger wie

```
int (*)[10] // Zeiger auf Vektor
```

Viel schöner wäre es doch, wenn man zusammengesetzte Datentypen mit einem Namen versehen und dann wie die Standardtypen verwenden könnte. Und genau das ist auch möglich, und zwar mit Typdefinitionen.

Eine Typdefinition wird mit dem Wortsymbol `"typedef"` eingeleitet. Rein syntaktisch sieht sie wie eine Variablendeklaration aus, wobei `"typedef"` als Speicherklasse, analog zu `"static"` oder `"extern"`, benutzt wird, nur daß damit eben keine Variable, sondern ein Datentyp deklariert Ein Beispiel:

```
typedef int Vektor[10];
```

Nun steht der Bezeichner `"Vektor"` für den Datentyp `"int[10]"` und kann fortan in Deklarationen und Typbeschreibungen aller Art benutzt werden, z. B.

```
Vektor vec, *tra, matrix[5];
typedef Vektor *Vektorzeiger;
void AddVec (Vektor v1, Vektor v2, Vektor ergebnis);
```

Die Bezeichnung „Typdefinition" ist dabei etwas irreführend, denn in Wirklichkeit definiert man keinen neuen Typ, sondern führt lediglich einen neuen Namen für einen Datentypen ein. Das ist keine Spitzfindigkeit, sondern führt z. B. dazu, daß die Deklarationen

```
void Ausgabe(Vektor v);
void Ausgabe(int v[10]);
```

ein und dieselbe Funktion bezeichnen, denn die jeweiligen Parameter haben denselben Datentyp, auch wenn sie unter unterschiedlichen Typbezeichnungen deklariert wurden.

Einem Bezeichner kann mehrfach derselbe Typ zugewiesen werden, beispielsweise

```
typedef char *Strvec[42];
typedef char *Strptr;
typedef char Zeichen;
typedef Zeichen *Strptr;      // Nochmal definiert, OK
typedef Strptr Strvec[42];    // auch OK
typedef Strptr Strvec[41];    // ERROR
```

Außer zur Abkürzung oder zur Steigerung der Übersichtlichkeit kann man Typdefinitionen auch zur besseren Portabilität von Programmen einsetzen. Unter MaxonC++ ist ein `"int"` bekanntlich 32 Bit breit, aber es gibt durchaus auch Systeme, bei denen das nicht so ist (z. B. sind im MessyDos-Sektor noch 16-Bit-ints üblich). Deshalb ist es empfehlenswert, in einem Programm am Anfang zuerst einmal alle Datentypen zu deklarieren, z. B.

```
typedef int i32;
typedef unsigned u32;
typedef short i16;
typedef unsigned short u16;
```

Wenn man im restlichen Programm dann nur noch seine selbstdefinierten Typbezeichnungen verwendet, braucht man bei der Übertragung auf andere Compiler nur noch diese Typdeklarationen anzupassen.

Dabei ist es naheliegenderweise nicht möglich, Symbole wie `"short"` oder `"unsigned"` auf selbstdefinerte Datentypen anzuwenden, also nach den obigen Definitionen folgendes zu schreiben:

```
unsigned i32 x;  // ERROR
u32 y;              // So ist's richtig
```

Eine Speicherklasse ist eine Eigenschaft eines bestimmten Datenobjekts, aber niemals Bestandteil eines Datentyps:

```
typedef static int SI;  // ERROR
```

Typqualifizierungen sind dagegen durchaus Teil eines Datentyps, womit ich wieder einmal elegant zum folgenden Abschnitt übergeleitet habe:

2.2.3 Konstante Daten auf der Flucht

2.2.3.1 Die „const"-Qualifizierung

Ein Feature, das zuerst in C++ eingeführt und dann von ANSI C abgekupfert wurde, ist die Typqualifizierung mit den Symbolen `"const"` und `"volatile"`. Man kann praktisch jeden Datentyp mit derartigen Qualifizierungen versehen und dann Objekte dieser Typen deklarieren, z. B.

```
const int i = 4711;
```

Der Wert der Variable `"i"`, die den Datentypen `"const int"` hat, kann (so gut wie) nicht verändert werden, bleibt also immer der, mit dem sie initialisiert wurde. Den Versuch einer Wertzuweisung an `"i"` würde der Compiler nicht durchgehen lassen. Dabei weist er sogar ziemlich clevere Versuche zurück, etwa den folgenden:

```
void main()
{  const int i = 4711;

   int *ip;
   ip = &i;          // ERROR
   *ip = 98;

   const int *ip2;
   ip2 = &i;
   *ip2 = 99;        // ERROR
}
```

Nimmt man die Adresse einer "const"-Variablen, z. B. mit "&i", ist das Ergebnis ein Zeiger auf "const int". Den Versuch, einen solchen Zeigerwert "ip2", also einem normalen Zeiger auf "int", zuzuweisen, läßt der Compiler nicht durchgehen, denn anschließend könnte man ja den Wert von "i" über diesen Zeiger verändern. Der zweite Versuch wird auch fehlschlagen. Zwar wird "ip2" als Zeiger auf "const int" deklariert, so daß die Adresse von "i" an "ip2" zugewiesen werden kann. Wenn dann aber bei "*ip2" der Inhaltsoperator darauf angewendet wird, kommt ein L-Wert des Typs "const int" dabei heraus, und Wertzuweisungen daran sind wieder einmal verboten.

Umgekehrt darf man einen Zeiger auf etwas Konstantes durchaus mit nicht-konstanten Daten initialisieren:

```
void f()
{ int i, *pi;
  const int ci, *pci;

  pci = &i;   // OK
  pi  = &ci;     // ERROR
}
```

Übrigens bezeichnet

```
const T*
```

tatsächlich einen Zeiger auf ein konstantes "T". Einen konstanten Zeiger auf "T" deklariert man als

```
T *const
```

Beides kann man natürlich auch kombinieren:

```
int i;

const int *const p = &i;  // Konst. Zeiger auf konst. "int"
```

Welchen Sinn hat nun dieses Gewurschtel mit dem "const"? Hauptsächlich dient dies dazu, anhand der Programmstruktur deutlicher auszudrücken, welche Daten wo verändert werden können. Beispielsweise kann man Vektoren nicht als solche, sondern nur in Gestalt von Zeigern darauf an Funktionen übergeben. Dadurch kann jede Funktion, die einen Vektor als Parameter hat, das als Argument übergebene Datenobjekt verändern, und man kann dann einem Funktionsaufruf nicht direkt ansehen, ob es dabei einen Seiteneffekt auf das Argument gibt. Sehen wir uns dagegen einmal die folgende kleine Funktion an, die nichts anderes tun soll, als einen String auszugeben:

```
#include <stream.h>

void Ausgabe (const char* st)
{ cout « st;
}
```

Der Programmierer kann innerhalb der Funktion `"Ausgabe"` nicht den Inhalt der Zeichenkette, auf die `"st"` zeigt, verändern, denn dann haut ihm der Compiler auf die Pfoten. Dann ist auch bei jedem Aufruf von `"Ausgabe"` klar, daß der Argument-String dadurch nicht verändert wird.

Die strengen Schutzvorschriften für konstante Daten gelten natürlich nicht nur für Zeiger, sondern auch für Referenzen. Eine Referenz auf einen nicht-konstanten Datentyp kann niemals mit der Adresse eines konstanten Objekts initialisiert werden, z. B.

```
const int i = 007;

int &ir1 = i;            // ERROR
const int &ir2 = i;   // OK
```

Rein syntaktisch ist es auch möglich, so etwas wie eine „konstante Referenz" analog zu den „konstanten Zeigern" zu deklarieren:

```
int &const irc = i;
```

In der Praxis hat man aber sowieso keine Möglichkeit, eine Referenz auf ein anderes Objekt als dem, mit dem sie initialisiert wurde, verweisen zu lassen, und deshalb ist es egal, ob man eine Referenz nun konstant deklariert oder in Peking eine Ente abstürzt.

Ein konstanter Referenz-Parameter wie

```
void f(const T &par);
```

ist eine echte Alternative zu „gewöhnlichen" Parametern wie

```
void g(T par);
```

vor allem dann, wenn `"T"` ein umfangreicher Datentyp ist, z. B. eine Struktur (siehe 2.5). Bei ersterem muß innerhalb der Funktion `"f"` bei jedem Zugriff auf `"par"` nachgesehen werden, wohin diese Referenz nun verweist, bei letzterem muß das Argument beim Funktionsaufruf auf den Stack kopiert werden. Der Programmierer sollte in solchen Situationen abschätzen, was wohl weniger Laufzeit kostet - im Zweifelsfall ist der Referenz-Parameter überlegen.

Der Datentyp

```
const char[20]
```

bezeichnet genaugenommen einen Vektor aus konstanten Zeichen und nicht einen konstanten Vektor. In der Praxis ist das aber natürlich dasselbe.

Eine „konstante Zeichenkette" der Länge `"n"` hat aus traditionellen Gründen den Typ `"char[n]"` und nicht etwa `"const char[n]"`. Das liegt daran, daß die allererste C-Version `"const"` noch nicht kannte und es deshalb prinzipiell erlaubt ist, den Inhalt einer konstanten Zeichenkette zu verändern (siehe auch 2.1.2.5.1). Heutzutage ist eine solche Definition natürlich nicht mehr besonders sinnvoll, aber man wollte eben zu älteren C-Implementierungen kompatibel bleiben.

2.2.3.2 Flüchtige Daten: „volatile"

Mancher Compiler ist ziemlich schlau, und das kann Probleme geben. Beispielsweise kann man auf dem Amiga ziemlich rüde die Tastatur abfragen, nämlich an der Speicheradresse `0xbfec01`:

```
void waitforspace()
{ unsigned char *key = (char*) 0xbfec01;

  while (*key != 0x7f);// Warte, bis "Space"-Taste gedrückt
}
```

Die Funktion wartet in einer `"while"`-Schleife darauf, daß die Leertaste gedrückt wird, indem immer wieder an der entsprechenden Speicheradresse nachgesehen wird, ob dort der entsprechende Code steht. Nun könnte ein Compiler ins Grübeln geraten und sich denken: „In dieser 'while'-Schleife wird ständig der Inhalt von `'*key'` benutzt, aber weder `'key'` noch `'*key'` oder eine andere Variable verändert. Also wäre es doch dumm, `'*key'` jedesmal wieder auszuwerten. Ich lade diesen Wert lieber am Schleifenanfang einmal in ein Prozessorregister und benutze dann immer den Registerinhalt."

Das Ergebnis wäre zwar prinzipiell eine relativ schlaue Optimierung, aber eben falsch: Der Compiler ahnt ganz einfach nicht, daß der Inhalt von `"*key"` sich auch ohne Zutun des Programms verändern kann. Für solche Fälle gibt es die Qualifizierung `"volatile"`, die völlig analog zu `"const"` eingesetzt werden kann. In der Funktion `"waitforspace"` ist die Variablendeklaration zu ersetzen durch:

```
volatile unsigned char *key = (char*) 0xbfec01;
```

Nun weiß der Compiler, daß `"key"` auf einen Wert zeigen soll, der sich ganz unabhängig vom Programm verändern kann (`"volatile"` heißt „flüchtig"), und daß er beim Ausdruck `"*key"` jedesmal wirklich an der jeweiligen Speicheradresse nachsehen muß.

Ganz penible Programmierer würden in diesem Fall übrigens `"const"` und `"volatile"` kombinieren:

```
volatile const unsigned char *key = (char*) 0xbfec01;
```

Aus der Speicherstelle `"0xbfec01"` kann man nämlich nur lesen, während Schreibzugriffe ergebnislos bleiben. Deshalb ist es für die (leider viel zu wenigen) Ästheten unter den C-Programmierern eine Selbstverständlichkeit, dieses dann auch durch ein zusätzliches `"const"` auszudrücken.

Da wir gerade bei solchem Brimborium sind: Noch schöner wäre natürlich

```
volatile const unsigned char *const key = (char*) 0xbfec01;
```

denn wir haben ja nicht das Bedürfnis, den Wert von `"key"`, also die Speicheradresse, noch einmal zu verändern. Nun könnte der Compiler wieder etwas optimieren: Er weiß jetzt, daß `"key"` immer auf die Adresse `"0xbfec01"` zeigen wird, und braucht fortan den Wert von `"key"` nicht mehr zu betrachten, sondern kann von dem konstanten Wert ausgehen.

Eine typische Anwendung für die "volatile"-Qualifizierung ist auch die Kommunikation zwischen Tasks. Auf dem Amiga kann man bekanntlich, wenn auch nicht ganz so einfach wie unter UNIX, mehrere Prozesse gleichzeitig laufen lassen. Die Kommunikation zwischen parallelen Prozessen könnte man der Einfachheit halber über gemeinsame Variablen laufen lassen, und genau diese Variablen muß man dann unbedingt als "volatile" deklarieren, denn sonst weiß der Compiler nicht, daß ein anderer Prozess sie hinterrücks verändern kann.

Im großen und ganzen ist "volatile" aber ein „Low level"-Konzept, daß ganz einfach den Compiler davon abhalten soll, da zu optimieren, wo es unerwünscht ist, wogegen die Qualifizierung "const" ein sehr mächtiges Sprachkonstrukt ist, das fernab jeder Maschinennähe oder Compilerabhängigkeit dabei hilft, Datenflüsse im Programmtext auszudrücken. Ein disziplinierter, auf stilistische Sauberkeit bedachter Programmierer wird "const" überall da benutzen, wo es sinnvoll ist, aber nur selten in eine Situation geraten, wo er "volatile" benutzen kann oder sogar muß.

2.2.4 Inline-Funktionen

Jeder Funktionsaufruf stellt einen ziemlichen Aufwand dar. Zuerst muß der aufrufende Code auf dem Stack Platz für die Argumente einrichten und diese dort ablegen. Dann folgt ein "JSR"-Sprung in die Funktion, die zunächst den nötigen Speicherplatz für ihre eigenen Daten auf dem Stack reserviert und die Inhalte aller Prozessorregister, die sie verändert, irgendwo abspeichert. Am Ende der Funktion werden dann wieder die alten Registerinhalte hergestellt, die lokalen Variablen werden vom Stack entfernt und mit "RTS" wird in das aufrufende Programm zurückgesprungen, wo dann zu guter Letzt auch noch die Argumente wieder vom Stack genommen werden müssen. Da gerät man als Programmierer bisweilen in Gewissenskonflikte: Soll man nun eine pisselige kleine Funktion definieren oder den Inhalt dieser Funktion jedesmal in das Programm einsetzen? Ein typisches Beispiel ist die Maximum-Funktion:

```
int max(int a, int b)
{ if (a>b)
     return a;
  else
     return b;
}
```

Statt eines Funktionsaufrufs "max(x,y)" könnte man bekanntlich alternativ jedes mal "(x>y: x : y)" einsetzen, was zwar schneller, aber bestimmt nicht lesbarer wäre. Für solche Fälle hat C++ ein nettes kleines Feature: „Inline"-Spezifikationen.

Wenn Sie vor die erste Deklaration (und auf Wunsch auch vor die folgenden) einer Funktion das Schlüsselwort "inline" setzen, wird bei jedem Aufruf dieser Funktion die Funktionsdefinition gewissermaßen in den aufrufenden Quelltext eingesetzt. Syntaktisch oder semantisch wird das Programm dadurch nicht verändert: Man deklariert einfach

```
inline int max(int a, int b)
{ if (a>b)
     return a;
```

```
  else
    return b;
}
```

und kann "max" dann wie eine ganz gewöhnliche Funktion benutzen. Der Unterschied liegt lediglich im dafür generierten Code. Mit "inline" wird der Inhalt von "max" jedesmal in den aufrufenden Code hineinkopiert. Dadurch wird so einiges gespart: Es muß weder mit "JSR" in die Funktion hinein- noch mit "RTS" aus ihr herausgesprungen, es müssen in der Regel keine Registerinhalte gesichert und nachher wieder hergestellt werden (es werden in der Kopie von "max" nach Möglichkeit nur solche Register benutzt, die im aufrufenden Code gerade frei sind), und die Funktion muß für ihre Daten keinen eigenen „Frame" auf dem Stack einrichten. Das Funktionsergebnis wird nicht notwendigerweise im Register d0, sondern dort, wo es gerade anfällt, zurückgegeben, und auch sonst hat der Compiler noch viele Optimierungsmöglichkeiten.

Eine Inline-Funktion wird also nicht nur einmal erzeugt, sondern in vielen Kopien, für jeden Aufruf eine. Es dürfte klar sein, daß Inlining Programme enorm aufblähen kann und man es deshalb damit nicht übertreiben sollte. Der Compiler darf aber prinzipiell das "inline" einfach ignorieren. MaxonC++ handhabt dies natürlich besser: Was "inline" sein soll, wird auch „geinlinet". Trotzdem kann es Situationen geben, in denen eine Inline-Funktion zusätzlich auf ganz gewöhnlich Art und Weise erzeugt werden muß, z. B. wenn man die Adresse einer solchen Funktion nimmt (siehe 2.3), wenn sie schon benutzt wird, bevor sie deklariert wird (schließlich kann der Compiler nicht hellsehen), oder wenn eine Inlinefunktion schlauerweise rekursiv ist (man kann eine Funktion ja nicht beliebig oft rekursiv in sich selbst hineinkopieren). All dies ist nämlich erlaubt, denn man kann eine Inline-Funktion ganz genau wie jede andere Funktion benutzen.

Wird eine Inline-Funktion aus einem der oben genannten Gründe (oder weil irgend so ein Compiler keine echten Inlines 'draufhat) real erzeugt, hat sie sinnvollerweise interne Linkage (wie mit "static" deklariert), so daß andere Module nichts davon merken, wenn eine Funktion in einer Übersetzungseinheit tatsächlich generiert wurde. Deshalb gelten für "inline" ähnliche Vorschriften wie für die Speicherklasse "static": Die Deklarationsreihenfolge

```
inline void f();
void f();
```

ist erlaubt,

```
void f();
inline void f(); // ERROR
```

hingegen nicht. Nebenbei bemerkt zählt "inline" nicht als Speicherklasse (wie "static" oder "extern"), sondern als „Funktions-Spezifierung" (wie auch "virtual", aber das kommt später). Diese Unterscheidung ist aber eher etwas für Standardisierungskommisionsmitglieder und Compilerbauer.

Abschließend ist als Zusammenfassung zusammenfassend zusammenzufassen, daß man mit "inline" zwar Laufzeit sparen kann, dies aber mit einer (oft erheblichen) Vergrößerung des Pro-

gramms erkaufen muß. Deshalb sollte man `"inline"` nur bei wirklich winzigen Funktionen einsetzen.

2.3 Zeiger auf Funktionen

2.3.1 Wohin zeigen Zeiger auf Funktionen?

Im Abschnitt 2.2.4 wurde bereits angedeutet, daß jede Funktion, einmal abgesehen von `"inline"`-Deklarationen, einen Code hat, der erzeugt wird und in den eingesprungen wird. Natürlich liegt dieser Code dann irgendwo im Speicher und hat folglich auch eine Adresse. Genaugenommen nimmt eine Funktion natürlich einen ganzen Speicherbereich ein, hat also eine Anfangs- und eine Endadresse. Für ihre Benutzung ist aber lediglich die Einsprungadresse interessant, im allgemeinen identisch mit der Startadresse. Eben diese charakteristische Adresse einer Funktion kann man „nehmen" und benutzen, um die Funktion auf dem Umweg über diese Adresse aufzurufen.

Wie gewohnt, müssen wir auch hier zunächst wissen, wie man einen Zeiger auf eine Funktion, bzw. eine solche Zeigervariable, deklariert. Sei `"f"` eine Funktion, die ein `"int"` als Parameter hat und `"double"` zurückgibt. Dann wissen Sie natürlich längst, wie man ein solches `"f"` deklariert:

```
double f(int);
```

Nun möge `"fp"` eine Variable sein, die auf eine Funktion eben dieser Art zeigt. Die deklariert man dann so:

```
double (*fp)(int);
```

Sie erinnern sich vielleicht: Die Klammern um `"*fp"` sind nötig, weil `"fp"` sonst eine Funktion ist, die einen Zeiger auf `"double"` zurückgibt, also im Prinzip das gleiche Theater wie bei dem „Zeiger auf Vektor" und „Vektor von Zeigern".

Weiter geht es dann ganz wie gewohnt: Man weist `"fp"` die Adresse einer passenden Funktion, also z. B. `"f"`, zu. Mit „passend" ist hier gemeint, daß Parameterliste und Ergebnistyp der Funktion mit dem von `"fp"` referierten Funktionstyp übereinstimmen müssen. Die Adresse einer Funktion nimmt unser bekannter Adressoperator `"&"`:

```
fp = &f;  // Nun zeigt "fp" auf Funktion "f"
```

Das Gegenteil vom Adress- ist der Inhaltsoperator, folglich kommen wir mit `"*fp"` an die Funktion, auf die `"fp"` zeigt, und können sie ganz normal mit einer geeigneten Argumentliste aufrufen, z. B. so:

```
double d1 = (*fp)(42);
```

Auch hier sind die Klammern um `"*fp"` nötig, denn sonst würde wieder einmal die Argumentliste stärker binden als der Inhaltsoperator: `"*fp(42)"` ruft eine Funktion `"fp"` auf (die es natürlich nicht gibt) und wendet auf deren Ergebnis den Operator `"*"` an.

2.3.2 Anmerkungen zu Syntax und anderen Details

Die Regeln, wie man mit Deklarationen wie

```
char*(*xxx)(char*);
```

klarkommt, wurden ja bereits in Abschnitt 2.1.2.3 genannt, und wenn Sie die beherzigen, werden Sie schnell herausfinden, daß hier `"xxx"` als ein Zeiger auf eine Funktion, die sowohl als Parameter und als Ergebnis einen Zeiger auf `"char"` hat, deklariert wird. Auch wissen Sie (hoffentlich) noch, daß man Datentypen wie Variablendeklarationen schreibt, nur daß man den Variablennamen wegläßt. Das soll hier, anhand eines Beispiels aus der Praxis demonstriert werden:

```
void konfus()
{ void *p;
  int a,b,c;

  (**(void(**)(...))p)(a,b,c); // Nanu?
}
```

Was ist das??? Also, wenn man die mysteriöse Zeile einmal betrachtet, stellt man fest, daß man auf oberster Ebene zwei aufeinanderfolgende Klammerausdrücke hat, deren rechter, `"(a,b,c)"`, nach allen C-Sprachregeln nur eine Argumentliste sein kann. Folglich ist der rätselhafte linke Teil der Anweisung ein Ausdruck, der einen Zeiger auf eine Funktion darstellt. Schaun' mer doch ma', was da so in der Klammer steht: Da sind zwei Inhaltsoperatoren `"**"`, ein weiteres Klammerpaar mit seltsamem Inhalt und schließlich der Bezeichner `"p"`. Dieser innere Klammerausdruck, `"(void(**)(...))"`, kann nur ein Cast sein, also eine explizite Typumwandlung, folglich muß `"void(**)(...)"` ein Datentyp sein. Wenn wir darin einfach einen Bezeichner einfügen,

```
void (**x)(...);
```

erhalten wir die Deklaration eines Zeigers auf einen Zeiger auf eine Funktion, die beliebige Parameter und Ergebnistyp `"void"` hat. Also wird in unserem Beispiel der Bezeichner `"p"` in genau jenen Typ verwandelt, anschließend wird zweimal der Inhaltsoperator angewandt, so daß wir die Funktion erhalten, auf die der Zeiger verweist, auf den `"p"` zeigt. Zum Abschluß wird diese Funktion mit den Argumenten `"(a,b,c)"` aufgerufen. Wenn man es so sieht, ist das doch ganz einfach...

Es gibt allerdings eine gewisse Konvention, nach der der Name einer Funktion identisch ist mit dem Zeiger auf Startadresse dieser Funktion, ähnlich wie der Name eines Vektors der Adresse des ersten Vektorelements entspricht. Deshalb kann man es sich in der Regel sparen, den Operator `"&"` auf Funktionen anzuwenden, z. B.

```
void f();

void main()
{ void (*fp)();

  fp = main;     // Statt "&main"
  fp = f;        // Statt "&f"
```

```
   (*fp)();        // Funktionsaufruf
}
```

In ANSI C ist sogar ausschließlich diese Schreibweise erlaubt, während C++ es dem Programmierer überläßt, ob er hier das `"&"` setzen will oder nicht.

Wenn aber der Name einer Funktion mit einem Zeiger auf diese Funktion identisch ist, besteht ein Funktionsaufruf immer aus zweierlei: einem Zeiger auf eine Funktion und einer Argumentliste. Deshalb ist es in Folge des obigen Features zusätzlich auch erlaubt, beim Funktionsaufruf über Pointer den Operator `"*"` wegzulassen:

```
void f();

void main()
{  void (*fp)() = f;

   fp();      // Statt "(*fp)()"
}
```

Dadurch sieht ein Funktionsaufruf über Zeiger syntaktisch wie ein gewöhnlicher Funktionsaufruf aus. Diesmal erlauben sowohl ANSI C als auch C++ beide Schreibweisen. Somit hängt es auch hier wieder von Ihrem persönlichen Geschmack ab, ob Sie ein `"*"` setzen wollen oder nicht.

2.3.3 Zeiger auf Überladene Funktionen

C++ kennt bekanntlich das schöne Feature des Überladens von Funktionen. Die daraus resultierenden Probleme beim Funktionsaufruf werden durch einen umfangreichen Satz von Matching-Regeln gelöst, so daß der Compiler anhand der Typen der Argumente stets die richtige Funktion herausfinden kann. Wenn man aber die Adresse einer Funktion nimmt, gibt man ja nur ihren Namen, aber keine Argumente an. Wie kann man also sauber angeben, welche Funktion nun bei Überladung gemeint ist? Dafür hat man in C++ eine verblüffend einfache Lösung gefunden.

Betrachten wir als Beispiel einmal die beiden Funktionen

```
int f(int);
double f(double);
```

Der Ausdruck `"&f"` (oder einfach `"f"`, ganz nach Geschmack) stellt nun nicht mehr einen Zeiger auf eine bestimmte Funktion dar, sondern hat den ganz eigenen Datentyp „Zeiger auf die überladene Funktion f" und keinen bestimmten Wert. Dieser Ausdruck kann aber in einen „passenden" Funktionszeigertyp umgewandelt werden, nämlich (klar) in die beiden Typen `"int (*)(int)"` und `"double (*)(double)"`. Einen Zeiger auf eine überladene Funktion kann man aber (im Gegensatz zu allen anderen Zeigertypen) nicht z. B. in `"void*"` umwandeln, denn dann kann der Compiler ja nicht aus dem Zusammenhang schließen, welche Funktion `"f"` gemeint ist.

Also enthält das folgende Programm zwei richtige und zwei falsche Initialisierungen:

```
int f(int);
```

```
double f(double);

void main()
{ int (*fp1)(int) = f;     // OK, da eindeutig
  void (*fp2)(int) = f; // ERROR, da kein passendes "f"

  void *vp1 = main;        // OK, "main" nicht überladen
  void *vp2 = f;           // ERROR, "f" ist überladen
}
```

Die von `"vp1"` referierte Funktion könnte man übrigens nicht direkt aufrufen, sondern muß den Zeigerausdruck erst casten, aber wie das geht, stand ja schon in 2.3.2.

2.3.4 Eine Anwendung: „qsort" und „bsearch"

Weniger erfahrene Programmierer werden jetzt vielleicht fragen, was das mit den Zeigern auf Funktionen eigentlich soll. Um es einmal ganz vornehm auszudrücken, sind Funktionszeiger eine elegante und schnelle Möglichkeit, nicht nur Daten, sondern auch Verfahren und Methoden abzuspeichern. Da der typische C-Programmierer aber eher ein praktisch denkender, bodenständiger Mensch ist, soll hier ein kleines Beispiel angeführt werden.

Es gibt Tage im Leben eines Programmierers, da muß man ganz allgemein Daten irgendwie sortieren. Die Algorithmenforschung hat im Laufe der Zeit etliche Sortierverfahren hervorgebracht, deren Darstellung ich mir hier aber ersparen möchte. Eines davon ist jedenfalls als „Quicksort" bekannt, und der Nomen, äh, Name ist ein Omen, denn dieses Verfahren ist wirklich ziemlich schnell. Als C-Programmierer müssen Sie es noch nicht einmal selbst implementieren, denn Quicksort steht als Standardfunktion zur Verfügung!

Dabei muß der Funktion irgendwie in Form eines Arguments gesagt werden, um welche Art von Daten es sich handelt und nach welchen Kriterien diese sortiert werden sollen. Genaugenommen braucht die Funktion folgende Angaben:

1. Wo liegt der zu sortierende Vektor (Anfangsadresse)?
2. Wie viele Elemente hat er?
3. Wie groß ist jedes einzelne Element? (Der genaue Datentyp ist egal, nur der Speicherplatzbedarf interessiert.)
4. Unter welchen Umständen soll ein Element im sortierten Vektor vor einem anderen stehen?

Die drei ersten Argumente kann man mit einem Pointer und zwei Zahlen erledigen, das letzte Argument kann sinnvoll nur in Form eines Zeigers auf eine Funktion realisiert werden.

Die Quicksort-Funktion ist in „*<stdlib.h>*" folgendermaßen deklariert:

```
void qsort(void *base, unsigned n, unsigned size, int (*cmp)(const
void*, const void*) );
```

Dabei bedeuten:

`"base"`: Zeiger auf Vektoranfang

`"n"`: Elementanzahl

`"size"`: Elementfröße

`"cmp"`: Zeiger auf eine Funktion, die zwei Vektorelemente vergleicht und einen negativen Wert liefert, wenn das erste Argument im Sinne der Sortierreihenfolge „kleiner" als das zweite ist, bzw. Null, wenn beide Elemente gleich sind, oder einen positiven Wert, wenn das erste Element „größer" als das zweite ist.

Als Beispiel wollen wir ganz einfach einen Vektor von hundert `"int"`-Werten sortieren und dann ausgeben. Um uns die Eingabe der Daten zu ersparen, ziehen wir einfach mit der Funktion `"rand()"`, ebenfalls aus `"<stdlib.h>"`, 100 Zufallszahlen.

Vor allem brauchen wir aber eine Vergleichsfunktion, die Zeiger auf zwei `"int"`-Objekte erhält und diese dann vergleicht. Formal muß man die Argumente leider als `"void"`-Pointer deklarieren:

```
int CompareInt(const void *p1, const void *p2);
```

Deshalb muß jede Benutzung der Parameter `"p1"` und `"p2"` mit einem Clast `"(int*)"` versehen werden. Zufälligerweise liefert eine Subtraktion von `"int"`-Werten gerade das gewünschte Ergebnis: negativ bei „kleiner", Null bei „gleich" und positiv bei „größer". Also sieht unser kleines Programm so aus:

```
#include <stdlib.h>   // Für "qsort" und "rand"
#include <stream.h>

int CompareInt(const void *p1, const void *p2)
{ return *(int*)p1 - *(int*)p2;
}

void main()
{ int Vek[100], i;

  for (i=0; i<100; ++i)   // Vektor initialisieren:
    Vek[i] = rand();        // Irgendwelche Zufallszahlen

  qsort(Vek, 100, sizeof(int), CompareInt);  // Sortieren

  // Daten ausgeben (10 Zahlen pro Zeile):
  for (i=0; i<100; ++i)
    cout « Vek[i] « (i%10 == 9 ? "\n" : " ");
}
```

Wozu ist ein sortierter Vektor gut? Einerseits ist es natürlich oft ganz einfach schöner, wenn man Daten sortiert ausgibt. Vor allem kann man Daten in einem sortierten Vektor wesentlich besser suchen. Stellen Sie sich z. B. einmal das Telefonbuch vor. Da die Einträge dort alphabetisch sortiert sind, muß man nicht das Ganze Buch von vorn an durchlesen, um einen Namen zu finden. Man schaut zuerst irgendwo in das Buch und stellt fest, ob der Eintrag, den man gerade gefunden hat,

vor oder hinter dem gesuchten steht, und sucht dann entweder vor oder hinter der aktuellen Position weiter. Dies wiederholt man dann so lange, bis man den gewünschten Eintrag gefunden hat.

Auch der Computer kann in sortierten Daten so suchen. Dabei wird der zu durchsuchende Bereich bei jedem Schritt halbiert, weshalb man dieses Verfahren „binäres Suchen" nennt. Auch dafür gibt es in `"<stdlib.h>"` eine Funktion, nämlich `"bsearch"`:

```
void *bsearch(const void *key, const void *base, unsigned n, unsigned
size,int (*cmp)(const void*, const void*));
```

`"key"` ist dabei ein Zeiger auf ein Datenobjekt, das im Vektor zu suchen ist, die übrigen Parameter sind wie bei `"qsort"`. In unserem Beispiel könnte man nach dem Sortieren mit

```
int j = 42;
void *position = bsearch(&j, Vek, 100, sizeof(int),
                         CompareInt);
```

prüfen, ob die Zahl `"42"` zufällig im Vektor vorkommt. Wenn ja, zeigt das Funktionsergebnis auf das entsprechende Vektorelement, und man könnte z. B. mit

```
(int*)position - base
```

den Index berechnen. Kommt `"42"` nicht vor, liefert `"bsearch"` das Ergebnis Null.

2.4 Aufzählungen

2.4.1 Deklaration

Die bis hier erwähnten Standard-Datentypen, also diverse Arten von Zahlen, Zeichen und Zeichenketten, sind zwar ganz nützlich, aber oft nicht ausreichend, um Daten zu repräsentieren. Wenn man die Daten einer Person abspeichern will (Achtung, immer brav das Bundesdatenschutzgesetz beachten!), können wir den Namen (eine Zeichenkette) oder das Geburtsjahr (ganzzahlig) mit den Standardtypen behandeln, aber wie drücken wir den Familienstand oder das Geschlecht in Daten aus?

Die Standardmethode dafür ist, eine numerische Kodierung einführen, etwa „0 heißt ledig, 1 verheiratet, 2 geschieden..." und solche Daten dann als Zahlen abzuspeichern. Schön ist das aber nicht gerade, denn diese nackten Zahlen sagen irgendwie nichts aus. Deshalb gibt es in C das Konzept der „Aufzählungstypen".

Eine Aufzählung oder „Enumeration" wird nicht aus den Standardtypen abgeleitet, sondern stellt einen ganz neuen, eigenen Datentyp dar. Deshalb kann man einen Aufzählungstypen nicht einfach beschreiben wie z. B. einen Pointertypen, sondern muß ihn deklarieren, und zwar zweckmäßigerweise unter einem bestimmten Namen.

Die Deklaration eines Aufzählungstyps besteht aus dem Schlüsselwort `"enum"` und dem (optionalen) Typnamen, z. B.

```
enum Farbe;
```

Mit so einer Deklaration kann man aber noch nichts anfangen, vor allem keine Objekte dieses Typs deklarieren. Jeder `"enum"`-Typ braucht sinnvollerweise eine Deklaration, bei der festgelegt wird, welche Werte er umfassen soll. Ein Beispiel:

```
enum Farbe { Rot, Gruen, Blau };
```

Der Aufzählungstyp `"enum Farbe"` umfaßt jetzt die drei Werte `"Rot"`, `"Grün"` und `"Blau"`. Jeder dieser Werte ist eine numerische Konstante, die durch einen Bezeichner repräsentiert wird. Das folgende Programmfragment demonstriert die Anwendung von Aufzählungen:

```
#include <stream.h>

enum Familienstand { ledig, verheiratet, geschieden, verwitwet };

void Heirat (enum Familienstand P1, enum Familienstand P2)
{ if (P1 == verheiratet || P2 == verheiratet)
    cout « "Ein Partner ist schon verheiratet!\n";
  else
    cout « "Geht in Ordnung.\n";
}

void main()
{ enum Familienstand f1, f2;

  f1 = ledig;

  // ...

  Heirat (f1, f2);

  // ...

  switch (f1)
  { case ledig:
      cout « "ledig";
        break;
      case verheiratet:
        cout « "verheiratet";          break;
      case geschieden:
      case verwitwet:
        cout « "nicht mehr verheiratet";
        break;
    }

  // ...
}
```

Man beachte vor allem, daß Aufzählungswerte Bezeichner und keine Strings sind. Folglich stehen sie im Programm ohne Hochkommata und können nicht direkt über `"cout"` ausgegeben werden.

Man kann auch Typ- und Variablendeklaraton vermischen:

```
enum Antwort {Ja, Nein} A1 = Ja, A2 = Nein, A3;
```

Diese Zeile deklariert den Aufzählungstyp `"Antwort"`, die beiden Konstanten `"Ja"` und `"Nein"` und die drei Variablen `"A1"`, `"A2"` und `"A3"`. Auf diese Weise ist es auch möglich (allerdings meist nicht sehr sinnvoll), namenlose Aufzählungstypen zu deklarieren:

```
enum {Ja, Nein} A1 = Ja, A2 = Nein, A3;
```

Der hier definierte namenlose Datentyp mit den beiden Werten `"Ja"` und `"Nein"` kann nicht noch einmal für Deklarationen benutzt werden: Wenn man eine Zeile wie

```
enum {Ja, Nein} A4;      // ERROR: "Ja"+"Nein" redefined
```

hinzufügt, beschwert sich der Compiler, daß die beiden Bezeichner `"Ja"` und `"Nein"` überdefiniert werden. Jede Aufzählung darf höchstens einmal definiert werden, auch nicht zweimal mit exakt denselben Werten.

2.4.2 Typnamen und Typlabel

Etwas seltsam ist, daß bei Aufzählungstypen in C das Schlüsselwort `"enum"` immer mitgeschleift werden muß: Nach der Deklaration

```
enum X {a, b, c,};
```

gibt es einen Datentypen namens `"enum X"`, während `"X"` allein kein Typname, sondern lediglich ein sog. „Typlabel" ist. Originellerweise kollidieren die Typlabel eines Scopes nicht mit anderen Bezeichnern, so daß folgende Deklaration durchaus möglich ist:

```
enum X {a, b, c};  // Typdefinition

enum X X;              // Variablendeklaration
```

Die erste Zeile deklariert einen Aufzählungstyp mit dem Typlabel `"X"`, während in der zweiten Zeile eine Variable namens `"X"` deklariert wird, die obendrein auch noch den Typ `"enum X"` hat... ganz schön verrückt!

Man kann dieses Feature aber auch benutzen, um das ewige `"enum"` zu vermeiden:

```
typedef enum X {a, b, c} X;
```

Auf diese Weise deklariert man sowohl ein Typlabel `"X"` als auch einen gleichnamigen Typbezeichner, und fortan kann man wahlweise mit

```
enum X x1;
```

als auch mit

```
X x2;
```

Variablen deklarieren - jedenfalls in ANSI C. In C++ haben Sie es wieder einmal einfacher: Nachdem man einen Aufzählungstyp deklariert hat, kann man das `"enum"` fortan weglassen. Ein Typlabel kann in C++ wie ein Typname verwendet werden. Es gilt aber trotzdem noch, daß gleichlautende Typlabel und Typnamen nicht kollidieren.

2.4.3 Aufzählungen und Zahlen

Naturgemäß kann ein Computer mit Zahlen viel besser umgehen als mit Bezeichnern oder Zeichenfolgen, und so verwundert es nicht, daß Aufzählungstypen intern als Zahlen dargestellt werden. In MaxonC++ ist ein Aufzählungstyp stets 32 Bit breit.

Ein originelles Feature ist, daß Aufzählungsdaten ohne explizite Konvertierung als `"int"`-Werte benutzt werden können:

```
enum Farbe { rot, gruen, blau };

int i = gruen;
```

Der Compiler zählt von Null an, also ist im obigen Beispiel `"rot" 0`, `"gruen" 1` und `"blau" gleich 2`, und die Variable `"i"` wird mit eins initialisiert. In der umgekehrten Richtung muß wieder einmal zwischen C und C++ unterschieden werden: In ANSI C darf ein ganzzahliger Ausdruck an eine Aufzählungs-Variable zugewiesen werden (wenn auch der Compiler eine Warnung ausgeben darf), in C++ geht das leider nicht, z. B.

```
int i = gruen;          // OK
enum Farbe f1 = i; // In C erlaubt, in C++ ERROR
```

Diese Einschränkung in C++ liegt daran, daß im allgemeinen nur wenige Zahlen gültige Werte eines Aufzählungstyps sind. Im Beispiel umfaßt der Typ `"Farbe"` nur drei verschiedene Werte, dargestellt durch die Zahlen `0` bis `2`. In C soll der Programmierer selbst darauf achten, daß er keine andere Zahlenwerte an eine `"Farbe"`-Variable zuweist, während C++ wieder einmal etwas pingeliger ist und derartige Versuche generell verbietet.

Entsprechend sieht es mit der Zuweisungskompatiblität zwischen verschiedenen Aufzählungstypen aus:

```
enum X {x1, x2} x;
enum Y {y1, y2} y = x;          // Nur in C, nicht in C++
```

In allen diesen Beispielen ist in C++ die Typwandlung mit einem Cast zu erzwingen:

```
enum Y {y1, y2} y = (Y) x;   // So geht's auch in C++
```

Manchmal will man, aus welchem Grund auch immer, die Zahlenwerte, die der Compiler den Aufzählungskonstante zuordnet, selbst bestimmen. Auch das ist möglich, indem man in der Typdefinition die gewünschten Werte angibt:

```
enum Farbe {rot = 1, gruen, blau = 4};
```

Wird einem Namen kein Wert zugeordnet, so wie hier `"gruen"`, zählt der Compiler einfach vom letzten vorher angegebenen Wert weiter, so daß `"gruen"` hier mit dem Wert `"2"` belegt wird. Die Konstante kann von jedem nach `"int"` wandelbaren Typ sein, also ein Ganzzahl- oder Aufzählungstyp. Es muß sich natürlich um einen konstanten, also schon zur Compilezeit berechenbaren Wert handeln.

Solche `"enum"`-Definitionen sind die „sauberste" Methode, in C ganzzahlige Konstanten zu deklarieren. Ein Beispiel:

```
enum { N = 100 };

int Vec[N];

void main()
{ for (int i=0; i<N; i++)
  Vec[i] = 0;

// usw.
}
```

Wenn Sie im obigen Programm die Anzahl der Vektorelemente ändern wollen, müssen Sie nur noch die `"enum"`-Deklaration entsprechend verändern. Zwar ist `"N"` formal keine ganze Zahl, sondern hat einen eigenen, namenlosen Aufzählungstyp, aber das macht in der Praxis nichts, denn Aufzählungstypen wie der von `"N"` können überall da eingesetzt werden, wo man auch `"int"` benutzten kann.

Dementsprechend kann man mit Aufzählungsdaten auch genau wie mit `"int"`-Zahlen rechnen:

```
enum E {a1, a2, a3, a4};

void main()
{ enum E e1 = a1;

  int i = e1+a2; // OK

  e1 += 1;          // ERROR
}
```

Der Ausdruck `"e1 += 1"` ist in C++ nicht erlaubt, denn er entspricht bekanntlich `"e1 = e1 + 1"`. Dabei wird die Summe `"e1 + 1"` aber sinnvollerweise als `"int"` ausgewertet, und eine Zuweisung der Form `"enum = int"` ist in C++ bekanntlich verboten.

2.5 Strukturen

2.5.1 Grundlagen

Es ist eine scheinbar triviale, aber in Wirklichkeit geradezu revolutionäre Idee, Daten, die zusammengehören, auch formal in einem Datentyp zusammenzufassen. Zum Beispiel besteht eine komplexe Zahl aus einem Realteil und einem Imaginärteil, die beide jeweils durch eine Fließkommazahl dargestellt werden können. Nun wäre es unschön, in einem Programm für jede komplexe Zahl gleich zwei Variablen zu deklarieren. Statt dessen definiert man sich einen Verbund-Datentyp namens `"Komplex"` (oder so ähnlich), der aus zwei Fließkommazahlen besteht. Das könnte man notfalls auch noch mit Vektoren realisieren:

```
typedef double Komplex[2];   // Element  [0] für Realteil,
                             //          [1] Imaginärteil
Komplex x1 = {1,0}, x2 = {2,2};
```

Aber hier haben wir ja auch das Glück, daß Real- und Imaginärteil mit demselben Datentyp, nämlich `"double"`, dargestellt werden können. Wenn wir aber Personendaten abspeichern wollen, und zwar Vorname (`"char[20]"`), Nachname (`"char[30]"`) und Geburtsjahr (`"int"`), geht das schon nicht mehr, denn naturgemäß haben alle Elemente eines Vektors den gleichen Typ. Genau hier setzt das Konzept der Datenstrukturen ein, das übrigens auch die Grundlage für die objektorientierte Programmierung darstellt.

Eine Struktur ist ein Datentyp, der Einzeldaten beliebiger Typen zusammenfaßt. Die Elemente einer Struktur werden nicht wie bei Vektoren mit einer Nummer, sondern mit einem Namen bezeichnet. Eine Struktur für die oben erwähnten Personendaten könnte man so deklarieren:

```
struct Person
{ char Vorname[20], Nachname[30];
  int Geburtsjahr;
};
```

Jetzt ist dem Compiler der Datentyp `"struct Person"` bekannt, und wir können uns Variablen deklarieren:

```
struct Person P1, P2;
struct Person *P_Zeiger = &P1;
```

und so weiter. Jedes Objekt des Typs `"struct Person"` besteht aus drei Datenelementen, die man in C++ auch als „Member" bezeichnet. Member werden formal genau wie Variablen deklariert. Der Zugriff auf einen Member erfolgt, indem man an einen geeigneten Ausdruck, z. B. einen Variablennamen, einen Punkt `"."` und den Namen des Members anhängt:

```
void main()
{ struct Person P;

  cout « "Bitte geben Sie ein:\n";
```

```
  cout « "Vorname:    "; cin » P.Vorname;
  cout « "Nachname:   "; cin » P.Nachname;
  cout « "Jahr:       "; cin » P.Geburtsjahr;
  cout « "\n";

  if (P.Geburtsjahr > 1992 || P.Geburtsjahr < 1880)
    cout « "Das Jahr ist offensichtlich nicht richtig!\n";
  else
  { int Alter = 1992 - P.Geburtsjahr;

    cout « "Hallo " « P.Vorname « " " « P.Nachname « ", Du bist " «
Alter «
              " Jahre alt.\n";
  }
}
```

Der Punkt `"."` zählt, genau wie ein Index oder eine Argumentliste, zu den Postfix-Operatoren und damit zur höchsten Bindungsklasse. Deshalb ist

```
*x.y
```

äquivalent zu

```
*(x.y)
```

und nicht etwa zu

```
(*x).y
```

Letzteres tritt übrigens so oft auf, daß es dafür eine Abkürzung gibt:

```
x->y
```

Um das ein wenig zu illustrieren, folgt auch prompt ein Beispiel:

```
struct S
{ char c1, *c2; }; // S hat zwei Member: char c1 und char *c2

void main()
{ struct S s1,s2, *sp;

  // ...

  sp = &s1;                // sp zeigt auf s1

  (*sp).c1 = *(s2.c2); // Ist äquivalent zu:
  sp->c1 = *s2.c2;
}
```

`"sp->c1"` ist der Member `"c1"` in der Struktur, auf die der Zeiger `"sp"` verweist, während `"*s2.c2"` das Zeichen ist, auf das der Member `"c2"` der Struktur `"s2"` zeigt.

2.5.2 Definitionen und Deklarationen

Die Deklaration einer Struktur beginnt, wie Sie sicher schon bemerkt haben, immer mit dem Wortsymbol `"struct"`. Anschließend ist alles optional, nur die Reihenfolge ist wichtig:

1. Strukturname, auch „Typlabel" genannt
2. Typdefinition, also die Deklaration der Member
3. Variablendeklarationen

Am Ende muß immer ein Semikolon stehen. Gehen wir nun halbwegs systematisch vor:

```
struct;
```

Diese Zeile ist ausnahmsweise nicht erlaubt, denn mindestens einer der drei oben angeführten Punkte muß folgen.

```
struct Name;
```

Diese Deklaration teilt dem Compiler mit, daß es einen Strukturtyp namens `"Name"` gibt. Die eigentliche Typdefinition kann später noch folgen, muß aber nicht. Nach einer solchen Mini-Deklaration kann man mit dem Typ `"struct Name"` nur solche Dinge tun, bei denen der Compiler nichts näheres über `"struct Name"` wissen muß. Man kann Zeiger auf diesen Typ deklarieren, z. B. `"struct Name *p;"`, aber keine Objekte dieses Typs selbst (`"struct Name s;"`). Man darf Funktionen deklarieren, die `"struct Name"` zurückgeben, aber keine Funktionen mit Parametern dieses Typs. Als kleines Extra darf man Objekte dieses Typs als `"extern"` deklarieren und somit importieren.

```
struct Name { int Memb1; char Memb2; };
```

Dies ist wohl die gebräuchlichste Form der Strukturdefinition. Jede Struktur darf beliebig oft deklariert werden (also so etwas wie `"struct Name;"`), aber höchstens einmal definiert werden. Man darf sie auch nicht zweimal absolut identisch definieren. Nach der Definition weiß der Compiler alles über den Strukturtyp, nämlich die Namen und Typen seiner Member und vor allem den Speicherplatzbedarf der Struktur. Die Deklaration

```
struct { int Memb1; char Memb2; };
```

ist zwar rein syntaktisch erlaubt, aber wenig sinnvoll. Hier wird ein Datentyp definiert, aber man kann nichts damit anfangen, denn er hat keinen Namen. Bei Strukturtypen zählt übrigens Namensgleichheit, d. h. zwei Strukturtypen mit absolut identischen Membern sind für den Compiler völlig unterschiedliche Datentypen.

```
struct Name objekt1, vektor[20], *zeiger;
```

Hier werden drei Variablen auf gewohnte Weise deklariert. Der Typ `"struct Name"` kann allgemein wie ein Standardtyp oder ein `"typedef"`-Bezeichner benutzt werden. Für die Deklarationen von `"objekt1"` und `"vektor[20]"` muß der Strukturtyp bereits definiert sein, die Pointervariable `"*zeiger"` kann auch schon vor der endgültigen Definition von `"struct Name"` erfolgen.

Man kann Strukturdefinition und Variablendeklaration auch kombinieren:

```
struct Name { int Memb1; char Memb2; } objekt1;
```

Diese Zeile deklariert gleichzeitig den Strukturtyp und eine Variable dieses Typs. Dabei kann man aber auch den Strukturnamen weglassen:

```
struct { int Memb1; char Memb2; } objekt1;
```

Auch hier werden ein Strukturtyp und eine Variable deklariert, aber der Strukturtyp hat keinen Namen und kann deshalb nie wieder für weitere Deklarationen benutzt werden.

Wenn Ihnen das andauernde `"struct"` auf die Nerven geht, können Sie eine Struktur auch so deklarieren:

```
typedef struct Name { int i; } TypName;
```

Fortan können Sie den Typ wahlweise `"struct Name"` oder einfach „TypName" nennen. Das kann man auch so weit treiben, daß die Struktur selbst keinen Typlabel mehr hat:

```
typedef struct { int i; } TypName;
```

Offenbar gibt es viele Ästheten oder Schreibfaule, die von Deklarationen wie der obigen reichlich Gebrauch machen. Deshalb sah man bei der Einführung von C++ Handlungs-

bedarf und legte fest, daß ein Typlabel in C++ wie ein `"typedef"`-Name benutzt werden kann, sofern es dabei nicht zu Kollisionen mit anderen Bezeichnern kommt:

```
struct Name { int i; };

Name objekt1;              // Nur in C++ erlaubt
struct Name objekt2;    // In C nötig, in C++ erlaubt
```

Die Deklaration der Member innerhalb einer Strukturdefinition ist formal identisch mit einer Variablendeklaration, außer daß man hier natürlich keine Speicherklassen verwenden kann (okay okay, in C++ können Member als `"static"` deklariert werden, aber das gehört hier noch nicht hin). Also können Member natürlich auch selbst wieder Strukturen sein, wie im folgenden Beispiel:

```
struct Datum
{ short int tag, monat, jahr; };

struct Name
{ char vorname[20], nachname[30];  };

struct Person
{ struct Name name;
  struct Datum geburt;
};

struct Person P1, P2, P3; // "struct" kann hier in C++
                                  // auch weggelassen werden
```

```
#include <string.h>   // Immer Ärger mit dem Stringshandling von C

void main()
{ P1.geburt.monat = 6;
  strcpy(P1.name.vorname,"Helmut");

  P2 = P1;                  // kopiert ganze Struktur
  P3.name = P1.name; // kopiert nur eine "Unterstruktur"

  // usw.
}
```

An diesem Beispiel sehen Sie auch, daß eine `"struct"` in C ein „echter" Datentyp ist, den man mit einer simplen `"="`-Zuweisung kopieren kann, während das bei Vektoren bekanntlich etwas krampfig ist (sieht man hier an `"strcpy"`). Man kann auch Strukturen direkt an Funktionen übergeben und nicht etwa bloß Zeiger darauf wie bei den Vektoren.

2.5.3 Initialisierung

Eine Struktur kann mit einer Struktur gleichen Typs initialisiert werden - einleuchtend, denn man kann solche Strukturen ja auch normal einander zuweisen. Daneben gibt es wie bei Vektoren die Möglichkeit einer Listeninitialisierung. Das ist nur bei Strukturen erlaubt, die keine Basisklassen, Konstruktoren oder private Member haben, aber da das alles Features sind, die in diesem Handbuch erst später behandelt werden, braucht Sie das vorerst nicht zu kümmern. Jedenfalls sind die Initialisierungswerte für die einzelnen Member in der richtigen Reihenfolge anzugeben und mit geschweiften Klammern `"{ }"` einzuschließen:

```
struct Anschrift
{ char Name[40], Anschrift[30];
  unsigned short PLZ;
  char Ort[25];
};

Anschrift A1 = {"Helmut Kohl",
                "Bundeskanzleramt",
                5300, "Bonn" };
```

Wenn man zu wenige Daten angibt, werden die übrigen Member mit Nullen initialisiert; bei zu vielen Daten beschwert sich der Compiler.

Bei verschachtelten Strukturen haben Sie wieder, wie bei mehrdimensionalen Vektoren, die Wahl, ob Sie Listen ineinander verschachteln oder nicht:

```
struct Datum
{ short int tag, monat, jahr; };

struct Name
{ char vorname[20], nachname[30];  };
```

```
struct Person
{ Name name;
  Datum geburt;
};

Person    P1 = { { "Karl", "Baumann" }, {26, 7, 1931} },
            P2 = { "Hein", "vom Deich", 24, 5, 66 };
```

Dasselbe gilt bei Vektoren von Strukturen:

```
struct S
{ char Name[20];
  int Jahr;
};

S Liste1[ ] = { {"Djikstra", 1940 }, {"Seville", 1976},
                    {"Davidson", 1903 } };

S Liste2[ ] = { "Dingenskirchen", 1952, "Werwohl", 1968,
                    "Nocheiner", 1967,"Schmuddel", 1972 };
```

2.5.4 Gültigkeitsbereiche und Scope-Regeln

Für die Typlabel von Strukturen gilt im Prinzip das gleiche wie bei `"enum"`-Typen: Sie haben einen eigenen Gültigkeitsbereich innerhalb jedes anderen Scopes. Dadurch sind Deklarationen wie

```
struct X X;
```

möglich: `"X"` ist der Name eines Objekts und `"struct X"` ein Typname, so daß Verwechslungen prinzipiell ausgeschlossen sind.

Aber jetzt kommt der Hammer: Da Strukturen und Klassen (was fast dasselbe ist) das zentrale Konzept der objektorientierten Features von C++ sind, hat man hier in C++ einige gravierende Änderungen gegenüber ANSI C vorgenommen. Zunächst einmal darf man, wie bereits unter 2.5.2 angedeutet, das Schlüsselwort `"struct"` (und auch `"class"` und `"union"`, dazu später mehr) weglassen und das Typlabel direkt als Typnamen verwenden, natürlich nur dann, wenn keine Kollisionen auftreten:

```
struct X { int x; };

X V;                // OK
struct X W;         // Das auch
X X;                // Sogar das geht!
X Y;                // Error
struct X Z;         // So ist's jetzt richtig
```

Ein vernünftiger Programmierer wird normalerweise Kollisionen von Typ- und Variablennamen vermeiden. Die Entwickler des Amiga-Betriebssystems gehörten offensichtlich nicht dazu, denn in den offiziellen Amiga-System-Includefiles findet man etliche derartige Kollisionen, z. B. muß man bei der

`"struct DateStamp"` aus *„libraries/dos.h"* das `"struct"` immer dazusetzen, weil in *„clib/dos_protos.h"* eine Funktion gleichen Namens deklariert wird... wirklich, ausgesprochen sauberer Stil!

Wirkliche Probleme gibt es mit der Kompatibilität zwischen C und C++ aber bei einem anderen Feature, den struktureigenen Gültigkeitsbereichen. Zu jeder Struktur gehört ein eigener Scope für die Namen der Member. Das bedeutet, daß die folgenden Deklarationen

```
struct S1 { int x; };
struct S2 { char x; };
double x;
```

nicht kollidieren: das erste `"x"` liegt im Gültigkeitsbereich von `"S1"`, das zweite im Scope von `"S2"` und das dritte auf Dateiebene. Es ist an jeder Stelle des Programms klar, welcher von den drei Bezeichnern mit `"x"` gemeint ist. Übrigens war das vor ANSI C nicht so direkt klar, denn man hatte ganz einfach vergessen, dies in den allerersten C-Spezifikationen festzulegen. Deshalb könnte es durchaus noch uralte C-Compiler geben, die obiges Programm nicht akzeptieren. Aber das kann Ihnen ja egal sein, denn Sie haben schließlich viel Geld für einen C++- und ANSI-C-Compiler hingeblättert und sind deshalb an derartigen Anachronismen nicht mehr interessiert.

Ach ja, ich sollte vielleicht mal langsam zur Sache kommen und verraten, wo denn nun hier der Unterschied zwischen C und C++ liegt. Also, weil Strukturen ja das Grundkonzept für die objektorientierte Programmierung darstellen, hat man ihnen in C++ einen ganzen Scope spendiert, während in ANSI C nur die Membernamen im Scope der Struktur liegen. Der Sinn dieser kryptischen Aussage wird wahrscheinlich am ehesten an einem Beispiel klar:

```
struct S1
{  int i;
   struct S2 *s2zeiger;     // VORSICHT!!!
};

struct S2
{  int j; };
```

Für den C-Compiler ist hier alles klar: Die Namen `"i"` und `"s2zeiger"` gehören in den Scope von `"S1"`, aber das Typlabel `"S2"`, das ja kein Membername, sondern eben ein Typlabel ist, gehört auf Dateiebene. Damit ist klar, daß der Member `"s2zeiger"` auf den Strukturtyp `"S2"` zeigt, der gleich anschließend definiert wird, und so ist das Ergebnis das erwartete.

In C++ sieht das anders aus: Hier kann man innerhalb einer Struktur nicht nur Member, sondern auch (aber das gehört schon zu den objektorientierten Features) Funktionen, Variablen oder eben auch Datentypen deklarieren, die dann lokal zur Struktur gehören und von außen normalerweise nicht sichtbar sind. Nun kommt also ein Compiler an und stellt fest, daß innerhalb von `"S1"` ein bisher völlig unbekannter Datentyp `"struct S2"` benutzt wird. Da der Datentyp neu ist, handelt es sich offenbar um eine Deklaration, und das Verhängnis nimmt seinen Lauf. Der Compiler nimmt gemäß der C++-Sprachdefinition an, daß es sich um einen zu `"S1"` lokalen Datentypen handeln muß. Wenn dann irgendwann später auf Dateiebene der Typ `"struct S2"` definiert ist, denkt der

Compiler nicht im Traum daran, daß dieses `"S2"` und das aus `"S1"` irgend etwas miteinander zu tun haben könnten. Die Folge: Der Member `"s2zeiger"` aus `"S1"` zeigt auf einen nicht definierten Typ, und

```
S1 s;
s.s2zeiger->j = 0;
```

führen zu einer Fehlermeldung, denn `"j"` ist ja Member des globalen `"S2"` und nicht etwa des lokalen.

Deshalb muß man in C++ stets folgendermaßen deklarieren:

```
struct S2;

struct S1
{ int i;
  struct S2 *s2zeiger; // Jetzt OK
};

struct S2
{ int j; };
```

Da `"S2"` jetzt zu dem Zeitpunkt, wo es in `"S1"` benutzt wird, bekannt ist, braucht der Compiler auch nicht mehr zu unterstellen, daß es sich um einen lokalen Datentyp handelt. Es gilt also wieder die Daumenregel, daß man in C++ alles erst möglichst sauber deklarieren muß, bevor man es benutzen kann.

Sie halten das hier diskutierte Thema für etwas esotherisch? Von wegen! Die Amiga-Standardincludes machen ausgiebig von solchen Pointereien zwischen Strukturen Gebrauch! Watt nu? Soll man bei jeder Benutzung dieser Includes vorher erst mit `"#pragma -"` in den C-Modus schalten, oder was?

Keine Panik: MaxonC++ gehört zu den relativ schlauen Compilern. Wird innerhalb einer Struktur ein anderer Strukturtyp deklariert und benutzt, aber nicht wirklich definiert, und außerhalb dieser Struktur ein Typ gleichen Namens definiert, so betrachtet MaxonC++ diese beiden Typen als identisch. Also schluckt MaxonC++ das Beispiel auch ohne vorherige Deklaration von `"struct S2"`. Durch dieses Feature gerät MaxonC++ natürlich in Konflikt mit dem gültigen Standard, und es kann passieren, daß unter MaxonC++ entwickelte Programme sich mit anderen Compilern nicht übersetzen lassen, aber andererseits ist dieses Feature durch Stroustrup abgesegnet (im „Anachronismen"-Kapitel des „Annotated C++ Reference Manual" von 1989, Abschnitt 18.3.5), und so erschien es uns sinnvoll, es zu implementieren.

2.5.5 Unionen

Unionen sind ein kleines Feature, mit dem man manchmal etwas Speicherplatz sparen kann. In C++ sind sie eigentlich überholt, denn Polymorphie läßt sich viel eleganter auf objektorientiertem Wege durch Vererbung erzielen. Aber wie so manche Altlast von C wurden auch Unionen in C++ aufge-

nommen, um die heilige Kuh namens „Kompatiblität" nicht ohne Not zu schlachten, und ungeachtet aller Vorbehalte sollen sie auch hier erwähnt werden.

Syntaktisch werden Unionen genau wie Strukturen behandelt, nur daß das Schlüsselwort `"struct"` durch `"union"` zu ersetzen ist. Der Trick ist dabei, daß alle Member einer union denselben Speicher belegen und man deshalb immer nur einen davon benutzen kann,

z. B.

```
union U { int Nummer; char *Name; };

U u1, u2;

void main()
{ u1.Nummer = 42;
  u2.Name = "Frood";
}
```

In diesem Beispiel muß also irgendwie inhärent klar sein, daß bei `"u1"` nur der Member `"Nummer"` und bei `"u2"` ausschließlich `"Name"` benutzt werden. Die jeweils anderen Member haben undefinierte Werte, und beim Versuch, sie auch noch zu initialisieren, werden die ersten Definitionen wieder überschrieben, denn `"Nummer"` und `"Name"` einer Struktur liegen ja im selben Speicher.

Damit dürfte ansatzweise deutlich geworden sein, daß man mit Unionen Speicherplatz sparen kann, wenn man von mehreren Membern immer nur jeweils einen pro Datenobjekt benötigt. Nehmen wir einmal an, wir wollen Daten von verschiedenen Computern abspeichern. Bei allen Computern wollen wir die Bezeichnung und die Speicherkapazität abspeichern, aber bei Amigas sollte man angeben, wieviel vom RAM CHIP-Mem ist, bei MS-DOSen sollte der Prozessortyp nicht verschwiegen werden, und bei ATARI-Computern braucht man unbedingt einen String mit einem Kommentar, was wir davon halten. Also entwerfen wir folgende Datentypen:

```
enum Typ {Amiga, MessyDOS, Atari};

struct Rechner
{ char Name[20];
  int kBytes;
  enum Typ wasistes;
  union
  { int ChipMem;
  char Prozessor[6];
    char *Kommentar;
  } Spezial;
};
```

Man beachte, daß wir hier einen namenlosen `"union"`-Typen innerhalb der Struktur definieren. Man hätte die Unionsdefinition auch herausziehen, auf Dateiebene mit einem Typlabel sauber definieren und dann innerhalb der Struktur benutzen können, aber so geht's natürlich auch. Anhand

des Members "wasistes" kann man jedenfalls jedem Datenobjekt ansehen, welcher Member der Union aktuell ist, z. B. so:

```
#include <stream.h>
#include <string.h>

void Ausgabe(Rechner &r)
{  cout « "Bezeichnung:   " « r.Name « "\n";
   cout « "Speicher:      " « r.kBytes « " kBytes\n";

   switch (r.wasistes)
   {  case Amiga:
         cout « "(davon " « r.Spezial.ChipMem « " kByte CHIP)\n";
         break;
      case MessyDOS:
         cout « "Prozessor:     " « r.Spezial.Prozessor « "\n";
         break;
      case Atari:
         cout « "Unsere Meinung dazu: "«r.Spezial.Kommentar«"\n";
         break;
   }
   cout « "\n";
}
```

Auch Unionen können mit „Listen" initialisiert werden, allerdings kann die „Liste" hier nur genau einen Initialisierungswert enthalten, und zwar für den ersten Member der Union:

```
void main()
{  RechnerR1 = { "Amiga 500", 3*1024, Amiga, { 512 } },
      R2 = { "ST", 1024, Atari },
      R3 = { "Taiwan Trashy 0815", 640, MessyDOS };

   // Bei "R2" und "R3" muß die "Spezial"-Information
   // nachträglich initialisiert werden:
   R2.Spezial.Kommentar = "Ach neee, muß ja nicht sein...";
   strcpy (R3.Spezial.Prozessor,"486sx");   // Hihihi...

   Ausgabe(R1);
   Ausgabe(R2);
   Ausgabe(R3);
}
```

Wo hier die Speicherplatzersparnis liegt? Eine Union braucht so viel Speicher wie ihr größter Member. Das "int" beim Amiga und "char*" beim Atari belegen jeweils 4 Bytes, während der String bei MessyDOS 6 Zeichen lang ist. Also ist die ganze Union 6 Bytes lang, was einer Ersparnis von 8 Bytes entspricht.

Zu guter letzt sei hier noch ein etwas abgefahrenes Feature von C++ erwähnt: namenlose Unionen. Wird eine Union ohne Typlabel einfach nur definiert, ohne daß eine Variablendeklaration o. ä. folgt, ist dies in Wirklichkeit eine Deklaration mehrerer Variablen, die alle an der gleichen Adresse liegen:

```
#include <stream.h>

static union
{ int i;
  char c;
  double d;
};

// Die drei globalen Variablen "i", "c" und "d" liegen
// alle an der gleichen Adresse! Der Beweis:

void main()
{ cout « int(&i) « "\n";
  cout « int(&c) « "\n";
  cout « int(&d) « "\n";
}
```

Das `"static"` ist hier aus rein formalen Gründen nötig, weil die Sprachdefinition das fordert. Eine solche Deklaration ist z. B. dann sinnvoll, wenn ein Programm mehrere große Vektoren hat, aber immer nur höchstens einen davon braucht. So etwas dürfte aber doch ziemlich selten vorkommen.

Namenlose Unions sind eher innerhalb von Strukturen nützlich:

```
enum Typ {Amiga, MessyDOS, Atari};

struct Rechner
{ char Name[20];
  int kBytes;
  enum Typ wasistes;
  union
  { int ChipMem;
    char Prozessor[6];
    char *Kommentar;
  };
};

Rechner R1;
```

Jetzt liegen die drei Member `"ChipMem"`, `"Prozessor"` und `"Kommentar"` nicht mehr in einer Union verborgen, sondern direkt in der Struktur `"Rechner"`. Also kann man sich auch das ewige `"Spezial"` sparen, z. B.

```
R1.Kommentar = "Ei der daus!".
```

Die Struktur `"Rechner"` sieht also noch genau so aus wie vorher, nur daß der Dummy-Member `"Spezial"` entfällt und dessen Member direkt in der Struktur stehen, und zwar auch wieder im selben Speicherbereich, der natürlich nicht mit dem von `"Name"`, `"kBytes"` oder `"wasistes"` kollidiert. Bemerkenswerterweise sind auch die Scope-Regeln dem angepaßt, d. h. die Bezeichner der Member der namenlosen Union könnten mit gleichlautenden Bezeichnern in der überliegenden Struktur zusammenrappeln, und das ist so ja auch sinnvoll.

Der Vollständigkeit halber sei hier schon darauf verwiesen, daß die meisten objektorientierten Features bei Unionen nicht unterstützt werden: Eine Union kann keine Basisklassen haben und keine Member, die einen Konstruktor oder Destruktor besitzen.

2.5.6 Bitfelder

Auch Bitfelder sind so ein Low-Level-Sprachkonstrukt, mit dem man bisweilen ein paar Byte sparen kann. Bekanntlich bietet C dem Programmierer ganzzahlige Datentypen in verschiedenen, für den jeweiligen Rechner typischen Breiten. Kein heute existierernder, ernstzunehmender Rechner unterstützt jedoch Zahlen von weniger als acht Bit, also einem Byte. Das liegt daran, daß der Zugriff auf den Speicher immer nur byteweise erfolgen kann, während das Lesen oder sogar Schreiben von einzelnen Bits einen recht hohen Aufwand darstellt.

Oft stellen 8-Bit-Werte aber schon eine Verschwendung dar. Ein typisches Beispiel sind Flags, also einfache boolesche Werte, die ohnehin nur die Werte `"0"` und `"1"` annehmen können, wofür ein Bit vollkommen reicht. Manchmal will man auch mehrere Zahlen, deren Wertebereich stark begrenzt ist, auf engstem Raum abspeichern, etwa in einem Langwort (32 Bit) vier Zahlen mit 20, 10 und zweimal je einem Bit unterbringen - die Alternative wäre eine Struktur aus einem `"int"`, einem `"short int"` und zwei `"char"`, die genau doppelt so viel Speicher benötigt.

Formal ist ein Bitfeld eine Struktur, bei der hinter jedem Member die Anzahl der benötigten Bits anzugeben ist, z. B.

```
struct Beispiel1
{ int a: 20, b: 10;
  unsigned int c: 1, d: 1;
};
```

Die Member eines Bitfelds dürfen ausschließlich ganzzahlige Datentypen haben. Es ist nicht erlaubt, Bitfelder mit normalen Strukturen zu vermischen, also für einige Member Bitbreiten anzugeben und für andere nicht. Ein Bitfeld kann nicht mit einer Elementliste initialisiert werden. Überhaupt ist vieles bei Bitfeldern implementationsabhängig und weder vom ANSI-C-Standard noch in der gängigen C++-Literatur verbindlich definiert. Der Zugriff auf einen Bitfeld-Member geschieht syntaktisch wie bei einer gewöhnlichen Struktur - aber Vorsicht, der Rechner benötigt dazu einen erheblich höheren Aufwand als bei einem gewöhnlichen Strukturzugriff. Das macht die Programme dann sowohl länger als auch langsamer. Deshalb sollte man es sich immer sehr genau überlegen, ob man wirklich Bitfelder einsetzen sollte. Oft wird dadurch der Programmcode dermaßen aufgebläht, daß die Speicherplatzersparnis bei den Daten wieder „ausgeglichen" wird, einmal ganz zu schweigen von der erheblich langsameren Zugriffsgeschwindigkeit auf die Daten!

Es ist erlaubt, hier namen- und typlose Member zu definieren:

```
struct BF2
{ int a: 3;
  :2;
  int b: 4;
};
```

Dadurch werden mitten zwischen `"a"` und `"b"` zwei Bits freigehalten. So etwas braucht man manchmal, wenn man ein vorgegebenes Layout übernehmen muß. Apropos Layout: MaxonC++ belegt die Bits von links nach rechts. Im obigen Beispiel belegt die Struktur `"BF2"` zwei Bytes, wobei die Bits 5 bis 7 des ersten Bytes `"a"` enthalten, die Bits 3 und 4 desselben Bytes unbelegt bleiben und `"b"` in dem drei untersten Bits des ersten sowie dem obersten Bit des zweiten Bytes liegt. Ein namenloser Member der Breite Null belegt nicht etwa Null Bits, sondern bewirkt eine Ausrichtung auf die nächste Bytegrenze:

```
struct BF3
{ int a: 3;
  :0;       // Macht das erste Byte "voll"
  int b: 8;
};
```

Dadurch liegt hier `"b"` genau in einem Byte, so daß ein Zugriff auf `"b"` zufällig keinen zusätzlichen Overhead erfordert.

Es ist nicht möglich, direkt oder indirekt oder wie auch immer die Adresse eines Bitfeld-Members zu nehmen:

```
struct S1 { int flag1:1, flag2:2 };

S1 s1;

int *p = &s1.flag1;      // ERROR
int &r = s1.flag2;       // ERROR
```

Generell kann ich nur raten, möglichst die Pfoten von Bitfeldern zu lassen, meist lohnt es sich ganz einfach nicht.

2.6 Dynamisch organisierte Datenstrukturen

2.6.1 Die Idee

Bisher haben wir alle Variablen, die ein Programm braucht, schon beim Schreiben des Programms deklariert. In der Praxis ist das allerdings keineswegs ausreichend, denn oft ist vorher nicht abzusehen, wieviel Daten das Programm zur Laufzeit benötigen wird. Nehmen wir an, Sie schreiben einen Editor und haben dabei eine Datenstruktur namens `"struct Zeile"`, die jeweils eine Zeile des Texts repräsentiert. Nun wissen Sie aber natürlich nicht, wie lang der Text sein wird, den der Benutzer einmal eingeben wird. Und genau hier kann man oft Hobby- und echte Programmierer unterscheiden: der Hobbyprogrammierer legt sich einen Vektor von, sagen wir einmal, zehntausend „Zeile"-Strukturen an und geht davon aus, daß der Benutzer nicht mehr als eben jene zehntausend Textzeilen eingeben wird, falls doch, hat er eben Pech gehabt. Andererseits belegt der Editor dann natürlich immer konstant einige Megabytes, aber das fällt ja nicht auf, schließlich hat man ja sowieso nur ein Programm gleichzeitig laufen. Auf dem Amiga geht das natürlich nicht, denn der Amiga-User wird sich „freuen", wenn ein Task den gesamten Speicher belegt.

Die einzig professionelle Methode, einen Editor zu programmieren, ist die benötigten Datenobjekte für Zeilen, Zeichen usw. bei Bedarf dynamisch zu erzeugen und wieder zu löschen, wenn sie nicht mehr benötigt werden, weil z. B. der Benutzer etwas aus dem Text gelöscht hat. Das beste Beispiel hierfür ist der in MaxonC++ verwendete Editor *EDWARD*. Und um solche Techniken soll es in diesem Abschnitt gehen, wobei wir auch einen eleganten Übergang zur objektorientierten Programmierung erhalten werden. Vorher sollten Sie sich allerdings vergewissern, daß Sie die Sache mit den Pointern auch wirklich verstanden haben, denn sonst wird es sehr, sehr schwer!

2.6.2 „new" und „delete"

Es soll also davon die Rede sein, wie man Datenobjekte zur Laufzeit des Programms dynamisch erzeugt und später auch wieder löscht. Sie waren es bisher gewohnt, daß ein Datenobjekt mit einer Variablen identisch ist und deshalb immer einen Namen hat. Ein dynamisch erzeugtes (man sagt neudeutsch auch „alloziertes") Objekt hat natürlich keinen Namen, denn dann müßte man es ja irgendwie im Programmtext deklarieren und hätte nichts gewonnen. Es liegt also nur irgendwo im Speicher 'rum und hat deshalb eine Adresse, und Adressen verarbeitet man in C mit Zeigern.

Rein praktisch erzeugt man in C++ ein neues Datenobjekt mit einem `"new"`-Ausdruck. Das Schlüsselwort `"new"`, gefolgt von einer Typbezeichnung, richtet an geeigneter Stelle im Speicher ein neues Datenobjekt dieses Typs ein und liefert als Ergebnis einen Zeiger darauf, z. B.

```
int *ip = new int;
```

Der Zeiger `"ip"` verweist jetzt also auf ein Datenobjekt `"*ip"` des Typs `"int"`, das vorher noch nicht existierte. Aber Vorsicht, es kann durchaus sein, daß nicht genug Speicher zur Verfügung stand! In diesem Fall liefert `"new"` den Wert `0`. Deshalb gehört hinter jeden `"new"`-Aufruf eigentlich eine `"if"`-Abfrage, die feststellt, ob auch wirklich Speicher reserviert wurde. Viele Programmierer sind leider zu faul dazu und vertrauen ganz einfach darauf, daß `"new"` klappt.

Aber auch für solche Leute gibt es in C++ ein nettes Gimmick: Im Include-File `"<new.h>"` wird eine Funktion namens `"set_new_handler"` deklariert, die als Argument eine `"void"`-Funktion ohne Parameter erwartet. Diese Funktion wird fortan automatisch aufgerufen, wenn eine Speicherreservierung mit `"new"` fehlschlug, und man kann in dieser Funktion versuchen, die Situation zu retten, oder doch wenigstens das Programm mit `"exit"` abbrechen.

Zum Löschen eines mit `"new"` erzeugten Objekts dient die `"delete"`-Anweisung:

```
delete ip;
```

Nun ist der Wert von `"ip"` laut Standard undefiniert und der Speicherbereich, in dem vorher `"*ip"` lag, wieder als unbelegt und frei definiert. In MaxonC++ zeigt `"ip"` nach dem `"delete"` übrigens immer noch auf dieselbe Adresse, aber es kann durchaus Implementationen geben, bei denen `"ip"` dabei auf `"0"` oder einen anderen Wert gesetzt wird. Wenn man versucht, ein Objekt zu löschen, das nicht mit `"new"` erzeugt wurde, steigt das Programm aus. MaxonC++ löscht am Programmende alle dynamischen Daten, die noch existieren.

Laut C++-Standard ist ein dynamisches Objekt nach seiner Erzeugung absolut undefiniert. MaxonC++ initialisiert den Speicher mit lauter Nullen, aber darauf kann man sich nicht generell verlassen.

Man kann Datenobjekte beliebiger Typen mit `"new"` erzeugen - mit einer Ausnahme, nämlich Referenzen. `"new int&"` liefert also einen Fehler, denn Referenzen sind in C++ strenggenommen keine Objekte, oder zumindest Objekte zweiter Klasse. Natürlich darf man auch keine Strukturen erzeugen, die noch nicht definiert sind, denn dann weiß der Compiler ja noch nicht, wieviel Speicherplatz dafür benötigt wird.

2.6.3 Dynamisch erzeugte Vektoren

Vielleicht fragen Sie sich jetzt schon, was der Quatsch soll. Eine Zeile wie `"int *ip = new int"` ist in der Tat ziemlich nutzlos, denn es kommt doch wohl auf dasselbe heraus, ob man eine `"int"`-Variable oder einen Zeiger darauf deklariert. Interessant werden dynamisch generierte Objekte erst dann, wenn man für eine Pointervariable jede Menge dynamischer Objekte alloziert und diese Anzahl vor der Ausführung des Programms noch nicht bekannt ist. Am einfachsten geht das in C++ über Vektoren, denn natürlich lassen sich auch solche Objekte dynamisch erzeugen:

```
void f()
{ char *cp;

  cp = new char[1000];
}
```

Man beachte, daß in C ein Vektor mit einem Pointer auf sein erstes Element identisch ist. Deshalb deklarieren wir hier `"cp"` als einfachen Zeiger auf `"char"` und nicht als Zeiger auf einen `"char"`-Vektor.

Nun ist auch das nicht so furchtbar schlau, denn man hätte hier einen entsprechend großen Zeichenvektor direkt deklarieren und sich die Sache mit dem Zeiger sparen können (einmal davon abgesehen, daß das hier erzeugte Objekt existiert, bis es gelöscht wird, also auch über die Ausführung der Funktion `"f"` hinaus). Praktischerweise akzeptiert C++ an dieser Stelle auch einen beliebigen variablen Ausdruck als Vektorgröße:

```
void g(int groesse)
{ char *cp = new char[groesse];
}
```

Na, ist das was? Das Programm kann jetzt erst einmal feststellen, wieviel Speicher es wohl brauchen wird, und dann einen entsprechend großen Vektor einrichten. Es gibt allerdings eine kleine Einschränkung: Bei mehrdimensionalen Vektoren darf ausschließlich der erste Indexbereich variabel sein:

```
void h(int i, int j)
{ new double[i][j];        // FALSCH
  new double[10][j];     // FALSCH
```

```
   new double[i][10];    // OK
}
```

Der Grund ist ganz einfach der, daß ein C-Programm beim Zugriff auf ein Vektorelement nicht wissen muß, wie viele Elemente der Vektor hat (der Programmierer soll gefälligst selbst darauf achten, daß er die Vektorgrenzen nicht überschreitet), wohl aber die Größe eines Vektorelements benötigt, denn alle Vektorelemente liegen im Prinzip hintereinander im Speicher, so daß man die Adresse des n-ten Elements aus der Vektoranfangsadresse erhält, indem man dazu das n-fache der Elementgröße addiert. Also stellt es für das Programm kein Problem dar, auf ein Element eines „beliebig großen Vektors von zehnelementigen double-Vektoren" zuzugreifen, denn ein „zehnelementiger double-Vektor" ist ganz einfach so groß wie zehn `"double"`-Zahlen. Umgekehrt kann er aber nicht auf ein Element eines „zehnelementigen Vektors von beliebig großen Vektoren von irgendwas" zugreifen, denn er weiß ohne weiteres nicht, wie groß ein `"double[j]"` ist und kann folglich mit Vektoren dieses Typs nichts anfangen.

Wie löscht man nun einen solchen Vektor variabler Größe wieder? Die Laufzeitbibliothek des C++-Systems merkt sich beim Allozieren eines Vektors variabler Größe irgendwo, wie groß der nun wirklich war. Beim Löschen ist dann hinter das `"delete"` ein Klammerpaar `"[ ]"` zu setzen:

```
void main()
{ int anz;

   unsigned *varivec = new unsigned[anz];  // ...

   delete [ ] varivec;
}
```

Bei älteren C++-Versionen („1.0-Standard") mußte man in diesen Klammern sogar die Vektorgröße angeben:

```
delete [anz] varivec;
```

Das ist heutzutage nicht mehr nötig. MaxonC++ akzeptiert es, um die Kompatibilität zu den alten Compilern zu wahren. Es ist absolut egal, was man in diese Klammern schreibt. Weglassen darf man diese Klammern aber nicht, sonst steigt das Programm wieder einmal aus.

2.6.4 Initialisierungen

Jede Variable kann initialisiert werden, folglich ist es nur recht und billig, wenn man dynamisch erzeugte Objekte ebenfalls initialisieren kann. Dabei ist der Initialisierungsausdruck beim `"new"` hinter den Datentyp zu setzen:

```
void fun(int *ip)
{ ip = new int(42);
}
```

Die Allozierungsanweisung ist also identisch mit

```
ip = new int;
*ip = 42;
```

Es gibt dabei allerdings ein paar Einschränkungen:

- Referenzen können nicht erzeugt und folglich auch nicht initialisiert werden.
- Vektoren können nicht initialisiert werden. Das kommt wohl daher, daß die Vektorgröße bei `"new"` variabel sein kann, C++ sonst aber nur Vektoren bekannter konstanter Größe initialisieren kann. Man müßte für diesen Fall völlig neue Regeln erfinden und implementieren, und das lohnt den Aufwand einfach nicht. Mit `"new"` erzeugte Vektoren fester Größe könnten zwar prinzipiell nach den bekannten Regeln initialisiert werden, aber so etwas kommt in der Praxis kaum vor (Sie erinnern sich, satt einen Vektor fester Größe zu allozieren, kann man ihn genausogut ganz normal deklarieren) und deshalb hat man darauf verzichtet.
- Strukturen können nicht mit Elementlisten initialisiert werden:

```
struct S { int a, b; };

S *sp = new S({1, 2});     // SYNTAX ERROR
```

Das erscheint auf den ersten Blick als ein großes Manko, aber Elementlisten sind ein altes, konventionelles C-Feature, während man Strukturen in C++ in der Regel mit „Konstruktoren" initialisieren wird. Da `"new"` ein neues Feature von C++ ist, hatte man wohl keine rechte Lust, hier auch noch diese antiquierten Elementlisten zu unterstützen.

Da ich gerade dabei bin, zu den objektorientierten Features vorzugreifen, will ich hier noch erwähnen,

- daß man beim Erzeugen einer Struktur mit einem „Konstruktor" stets geeignete Argumente dafür als Initialisierung angeben muß, es sei denn, die Struktur hat einen „Default-Konstruktor",
- daß ein Vektor von Strukturen nur dann mit `"new"` alloziert werden kann, wenn die Struktur einen „Default-Konstruktor" besitzt, welcher dann für jedes einzelne Vektorelement aufgerufen wird, und
- daß beim Löschen einer Struktur mit `"delete"` ggf. der „Destruktor" dieser Struktur aufgerufen wird. Entsprechend wird bei Vektoren von Strukturen der Destruktor für jedes einzelne Vektorelement aufgerufen.

Aber von all' diesen geheimnisvollen Features gehen wir jetzt zuerst noch einmal zurück in das finstere C-Mittelalter:

2.6.5 „malloc" und andere Antiquitäten

Wie bereits erwähnt, hatten in alten Tagen die C-Programmierer (die es heute angeblich immer noch geben soll, aber das ist bestimmt nur eine Legende) ein hartes Leben: ANSI C kennt weder

"new" noch "delete". Statt dessen lief die Speicherverwaltung im wesentlichen über zwei Funktionen aus "<stdlib.h>". Man reservierte sich Speicher mit der Funktion

```
void *malloc(size_t size):
```

Dabei bezeichnet "size_t" einen Datentypen, der ebenfalls in "<stdlib.h>" deklariert wird und gerade der Typ ist, den "sizeof" als Ergebnis hat (in MaxonC++ ist das "unsigned int"). Die Funktion "malloc" reserviert jedenfalls "size" Bytes und gibt einen Zeiger darauf zurück:

```
#pragma -
#include <stdlib.h>

void main()
{ int *ip = malloc(sizeof(int));
  /* usw. */
}
```

Man beachte, daß man in C++ nicht so ohne weiteres ein "void*" in einen anderen Pointertyp verwandeln kann, weshalb man in C++ einen Cast setzen müßte:

```
int *ip = (int*) malloc(sizeof(int));
```

Aber es wird wohl niemand mehr in C++ noch "malloc" verwenden.

Man kann im allgemeinen nicht erwarten, daß ein "n"-elementiger Vektor gerade "n"-mal so groß ist wie sein Elementtyp. Die Struktur

```
struct S { int i; char c; };
```

z. B. nimmt in MaxonC++ gerade fünf Bytes ein, nämlich vier für "int" und eins für "char". Auf dem MC68000 und vielen anderen Prozessoren auch müssen Langwortdaten aber immer an einer geraden Adresse stehen. Deshalb wird der Compiler bei einem Vektor von "S" für jedes Element ein zusätzliches Füllbyte reservieren, so daß jedes Vektorelement und damit auch das darin enthaltene "i" an einer geraden Adresse liegt. MaxonC++ fügt hier sogar drei Bytes ein, um eine Langwort-Ausrichtung zu erzwingen. Langer Rede, schwacher Sinn: Es ist nicht empfehlenswert, mit "malloc" einen Vektor variabler Größe einzurichten, denn

```
malloc(n * sizeof(struct S))
```

führt aus den oben angeführten Gründen oft zu falschen Ergebnissen, und

```
malloc(sizeof(struct S[n]))
```

ist in C ganz einfach falsch, denn hier gibt es keine Vektoren nicht-konstanter Größe. Also gibt es auch noch die Funktion

```
void *calloc(size_t nobj, size_t size)
```

zum korrekten Allozieren von Vektoren, wobei der erste Parameter die Anzahl der Elemente und der zweite die normale Größe des Elementtyps darstellt:

```
struct S *var_s_vec = calloc(n, sizeof(struct S))
```

"calloc" kennt die „Alignment"-Regeln des jeweiligen C-Systems und kann deshalb die notwendige Anzahl von Füllbytes einfügen. "malloc" steht wohl für „Memory ALLOCation", aber was das "c" am Anfang von "calloc" heißen soll, weiß ich auch nicht. Mir persönlich würde der Name "colloc" viel besser gefallen, denn das ist wenigstens ein Palindrom.

Zu guter Letzt braucht man natürlich auch noch eine Funktion, die Speicher wieder freigibt:

```
void free(void *p);
```

Man übergibt hier wie bei der "delete"-Anweisung einen Zeiger auf das zu löschende Objekt, das entweder mit "malloc" oder "calloc" erzeugt worden sein muß, z. B.

```
free(var_s_vec);
```

Man hat hier also keine unterschiedlichen Freigabeanweisungen wie in C++ ("delete" und "delete[]").

Man sieht sofort, daß diese C-Funktionsaufrufe nicht gerade besonders schön aussehen, und da man in C++ ohnehin dabei war, C gründlich zu erweitern, hat man die viel schöneren "new"- und "delete"-Ausdrücke eingeführt. "new" und "delete" sind übrigens zwei Wortsymbole und damit ein echter Bestandteil der Programmiersprache C++, während "malloc" und "free" keine wirklichen C-Sprachkonstrukte, sondern nur zwei Bibliotheksfunktionen wie andere auch sind. Das Konzept der dynamischen Datenorganisation wurde in C++ also „aufgewertet".

Neben diesen ästhetischen Überlegungen gibt es aber auch noch ganz handfeste Gründe, weshalb "malloc" und "free" für objektorientierte Programmierung ungeeignet sind: Der C++-Compiler muß aus Gründen der Konsistenz eine Möglichkeit haben, festzustellen, ob ein neues Datenobjekt erzeugt bzw. ein bestehendes gelöscht wird, weil dabei eventuell Konstruktoren oder Destruktoren aufgerufen werden müssen.

Für alle, die immer noch am sittlichen Nährwert der dynamischen Datenorganisation zweifeln, und insbesondere alle die, die glauben, daß sie so etwas nie kapieren werden, möchte ich zu diesem Thema ein etwas größeres Beispiel bringen:

2.6.6 Beispielprogramm: Eine Liste von Daten

2.6.6.1 Das Problem: Ein wirrer Haufen von Daten

Ein Problem, auf das man als Programmierer immer wieder stößt, sind Listen von irgendwelchen Daten. Anfänger, lösen so etwas mit Vektoren, was zwar einfach und auch schnell ist, aber den ganz gewaltigen Nachteil hat, daß man normalerweise Vektoren fester Größe deklarieren muß. Auch Vektoren variabler Größe (siehe 2.6.3) sind nicht so ganz das Gelbe vom Ei, denn hier muß man schon in dem Moment, in dem die Datenstruktur erzeugt wird, wissen, wie viele Elemente die Liste höch-

stens umfassen soll. Nein, für echte Programmierer führt wirklich kein Weg an dynamischen Listen vorbei.

Die lineare Liste ist bestimmt nicht die beste, aber die einfachste und deshalb gebräuchlichste Datenstruktur. Auch das Amiga-Betriebssystem macht intern von solchen Strukturen ausgiebig Gebrauch. Was versteht man aber unter einer Liste?

Als Beispiel wollen wir ein kleines Telefonbuch erzeugen. Jeder Eintrag umfaßt hier zunächst einen Namen und eine Telefonnummer:

```
struct Eintrag_erster_Entwurf
{ char Name[30];
  char Nummer[20];
};
```

Nun stellen wir an unsere Datenstruktur aber folgende Anforderungen:

1. Die Anzahl der Datensätze ist ausschließlich durch das zur Verfügung stehende RAM begrenzt. Dabei soll die Struktur zu keinem Zeitpunkt mehr Speicher als unbedingt erforderlich benötigen.
2. Die Datensätze können in beliebiger Reihenfolge eingegeben werden und werden dabei sofort sortiert.
3. Daten können aus der Struktur gelöscht werden. Dabei können Einfüge- und Löschoperationen in beliebiger Reihenfolge erfolgen.

Und alles das soll auch noch halbwegs schnell vor sich gehen. Wie packt man nun ein solches Problem an?

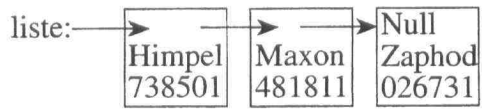
Es führt offensichtlich kein Weg daran vorbei, die Daten dynamisch im RAM einzurichten. Die Idee der Liste ist, daß jedes Element einen Zeiger auf das nachfolgende besitzt, etwa so:

```
struct Eintrag_erster_Entwurf
{ Eintrag_erster_Entwurf *next;
  char name[30];
  char nummer[20];
};
```

Alles, was wir jetzt noch brauchen, ist ein Zeiger auf das erste Listenelement:

```
Eintrag_erster_Entwurf *liste;
```

Der Name des ersten Listenelements ist dann wie gewohnt `"liste->name"`, der des zweiten Elements `"liste->next->name"`, an das dritte Listenelement gelangt man über `"liste->next->next->name"` und so weiter. Natürlich muß jede Liste ein Ende haben, und das markiert man normalerweise, indem man dort als `"Next"`-Zeiger eine Null einträgt. Ein „Telefonbuch" mit drei Einträgen könnte also folgendermaßen aussehen:



Es dürfte anschaulich klar sein, wie man sich mit einem Zeiger durch eine solche Struktur hangeln kann, aber nur in eine Richtung, vom Anfang zum Ende. Man nennt eine solche Struktur eine „einfach verkettete Liste".

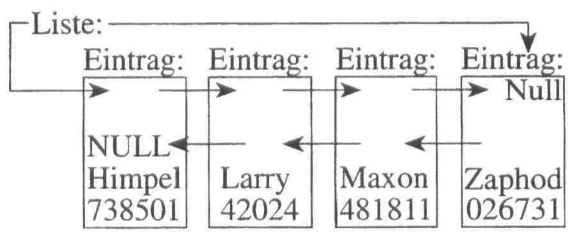
2.6.6.2 Die Lösung: Eine doppelt verkettete Liste

Etwas komplizierter ist eine doppelt verbundene Liste, mit der wir uns im folgenden befassen wollen. Dabei hat jedes Listenelement gleich zwei Zeiger, wie gehabt einen auf den Nachfolger und zusätzlich einen auf den Vorgänger:

```
struct Eintrag
{  Eintrag *next, *prev;

   char name[30];
   char nummer[20];
};
```

Auch jetzt reicht es eigentlich, wenn wir einen Zeiger auf den Listenanfang haben, aber da wir schon die Möglichkeit haben, rückwärts durch die Struktur zu laufen, benutzen wir hier aus Symmetriegründen lieber gleich zwei Zeiger: Einen auf das erste und einen auf das letzte Element der Liste. Das schreit wiederum danach, diese beiden Zeiger in einer neuen Struktur namens `"Liste"` zusammenzufassen:

```
struct Liste
{  Eintrag *anfang, *ende;
};
```



Am besten macht man sich auch das wieder mit einem „Bildchen" klar, diesmal gleich mit vier Einträgen:

Das sieht ja schon ganz schön kompliziert aus, und wir haben es nun auch mit zwei verschiedenen Datentypen zu tun. Aber keine Angst, wenn man sich erst einmal an das Listen-Konzept gewöhnt hat, ist das ganz einfach - ehrlich!

Zum „Aufwärmen" schreiben wir uns eine Funktion, die alle Daten einer Liste ausgibt. Wir nennen sie folgerichtig `"Ausgabe"` und als Parameter erhält sie eine Referenz auf eine `"Liste"`. Wir benutzen hier besser eine Referenz, weil dann beim Funktionsaufruf nur ein Zeiger auf die Liste auf den Stack kopiert werden muß, während sonst die ganze Struktur (immerhin zwei Zeiger) kopiert werden müßte. Weil wir aber nicht vorhaben, bei der Ausgabe irgend etwas an der Liste zu verändern, benutzen wir die Qualifizierung `"const"`:

```
void Ausgabe (const Liste &L)
{
```

Wie geht's jetzt weiter? Wir müssen irgendwie durch die ganze Liste laufen, und zwar vom Anfang bis zum Ende. Also brauchen wir einen Zeiger auf ein „Element", den wir gleich auf das erste Element der Liste setzen:

```
Eintrag *E = L.anfang;
```

Jetzt müssen wir folgende drei Schritte iterieren:

1. Prüfe, ob das Listenende erreicht ist; falls ja, sind wir fertig.
2. Andernfalls gebe das Listenelement, auf das `"E"` zeigt, aus.
3. Setze `"E"` auf das nächste Listenelement und mach bei (1.) weiter.

Genau das besorgt die folgende Schleife:

```
while (E != 0)
{ cout « E->name « ": " « E->nummer « "\n";
  E = E->next;
}
```

Hier gibt es noch einiges zu verbessern, aber wir wollen es einmal dabei belassen. Die komplette Funktion sieht also so aus:

```
#include <stream.h>

void Ausgabe(const Liste &L)
{ Eintrag *E = L.anfang;

  while (E != 0)
  { cout « E->name « ": " « E->nummer « "\n";
    E = E->next;
  }
}
```

Man könnte sie aber auch so schreiben:

```
void Ausgabe (const Liste &L)
{ for(Eintrag *E = L.anfang; E; E = E->next)
  cout « E->name « ": " « E->nummer « "\n";
}
```

Durch scharfes Hinsehen werden Sie feststellen, daß es sich hier nur um eine andere Schreibweise desselben Algorithmus handelt. C ist eben eine sehr „flexible" Sprache - man kann Programme beliebig konfus schreiben. „But now to something completely different:"

2.6.6.3 Initialisieren und Einfügen

Auch eine Liste muß initialisiert werden, nämlich als leere Liste. Am besten schreibt man sich auch dafür eine kleine Funktion:

```
void Init(Liste &L)
{ L.anfang = 0;
  L.ende   = 0;
}
```

Das Schöne an C++ ist ja, daß man Funktionen überladen kann. Sie können also in einem Programm -zig Datentypen haben und für jeden so eine Funktion namens `"Init"` definieren, der Compiler weiß, welche er zu nehmen hat. Noch nicht einmal das müssen Sie tun, denn dafür gibt es Konstruktoren und Destruktoren, aber das gehört jetzt nicht hier hin.

Eine leere Liste ist nicht besonders aufregend, also brauchen wir dringend eine Funktion, die einen Datensatz in die Liste einfügt. Auch dafür schreiben wir eine Funktion, die als Parameter eine Listenstruktur, einen Namen und eine Telefonnummer erhält. Die Funktion liefert einen Zeiger auf das neu erzeugte Listenelement zurück. Wenn kein Speicher reserviert werden konnte, gibt sie eine Null zurück.

Also deklarieren wir unsere Funktion - aber nicht, ohne vorher die Stringfunktionen einzubinden:

```
#include <string.h>

Eintrag *Einfueg(Liste &L, const char Name[], const char Nummer[])
{
```

Es geht jetzt ziemlich simpel los: Wir erzeugen ein neues Element und geben eine Null zurück, wenn das fehlschlug:

```
Eintrag *neu = new Eintrag;

if (!neu)
   return 0;
```

Jetzt sollten wir das neue Element noch mit den Argumenten der Funktion initialisieren, wobei wir wegen dieses von C geerbten *%&@ß-Stringhandlings auf die `"strcpy"`-Funktion zurückgreifen müssen:

```
strcpy(neu->name, Name);
strcpy(neu->nummer, Nummer);
```

So, nun wird es kompliziert. Als erstes müssen wir herausfinden, wo wir das neue Element in der Liste einhängen sollen. Deshalb lassen wir einen Zeiger namens `"pos"` durch die Liste laufen, bis

wir ein Element finden, dessen "name" alphabetisch größer als der neue ist, wobei allerdings der Sonderfall auftreten kann, daß es kein solches Element gibt:

```
Eintrag *pos = L.anfang;

while (pos!=0 && strcmp(pos->name, Name) < 0)
  pos = pos->next;
```

Man beachte, daß C logische Ausdrücke grundsätzlich minimal auswertet: Wenn die Klausel "pos != 0" schon nicht erfüllt ist, wird der zweite Teil der Schleifenbedingung, nämlich das "strcmp", gar nicht mehr ausgewertet.

Was heißt das nun, wenn die Schleife mit "pos == 0" anhält? Offensichtlich ist das neue Element das größte der ganzen Liste, was aber auch daran liegen kann, daß die Liste ganz einfach leer ist. Also testen wir zuerst diese Sonderfälle ab:

```
if (pos == 0)
// Sonderfall: Anfügen an das Ende der Liste
{ Eintrag *alt = L.ende;  // Altes Listenende
```

Der Zeiger "alt" zeigt also auf das bisher letzte Element der Liste. Wenn die Liste bisher leer war, ist "alt" natürlich Null, andernfalls:

```
if (alt != 0)
// Allgemeiner Unter-Fall: wirklich anhängen
{ alt->next = neu;
  neu->prev = alt;
  neu->next = 0;
  L.ende = neu;
}
```

Langsam zum Mitdenken: Zuerst setzen wir "neu", also das neue Listenelement, als Nachfolger von "alt", also dem bisherigen Listenende, und anschließend definieren wir umgekehrt "alt" als Vorgänger von "neu". Nun müssen wir nur noch den Nachfolgerzeiger des neuen Elements auf Null setzen, denn es ist nun ja das letzte Listenelement, und den Eintrag "ende" der Listenstruktur aktualisieren, indem wir ihn auf das neue Element umhängen. Wenn Sie das jetzt nicht verstanden haben, nehmen Sie sich am besten einen Bleistift und die Rückseite eines gebrauchten Briefumschlags und malen sich die ganze Sache auf.

Es fehlt noch das "else" zum zweiten "if", also der Fall, daß sowohl "pos" als auch "alt" sind Null. Das kann nur daran liegen, daß unsere Liste noch völlig leer ist:

```
else
// Spezieller Spezialfall: Liste ist noch leer
{ neu->next = 0;
  neu->prev = 0;
  L.anfang = neu;
  L.ende = neu;
}
```

Diese Zuweisungen dürften halbwegs verständlich sein: Das neue Element ist das einzige und hat deshalb weder Vorgänger noch Nachfolger. Es ist jetzt gleichzeitig das erste und das letzte Listenelement, weshalb beide Zeiger der Listenstruktur auf das neue Element gesetzt werden müssen.

Nun poppen wir eine Ebene 'rauf und überlegen uns, was wir tun, wenn `"pos"` nicht auf Null zeigt, wir also ein Element gefunden haben, vor das wir das neue hängen müssen. Hier gibt es dann gleich noch einen Sonderfall: Möglicherweise ist das neue Element das kleinste der ganzen Liste und muß deshalb an den Anfage gehängt werden. Das erkennt man wahlweise daran, daß `"pos == L.anfang"` gilt oder daß `"pos"` keinen Vorgänger hat:

```
  }
  else
  if (pos->prev == 0)
  // Sonderfall: Neues Element an Listenanfang hängen
  { pos->prev = neu;
    neu->next = pos;
    neu->prev = 0;
    L.anfang = neu;
  }
```

Das geht völlig analog zum Fall „Anhängen an das Listenende", nur daß wir hier mit `"pos"` schon einen Zeiger auf das bisher erste Listenelement haben und deshalb keine Hilfsvariable wie `"alt"` einführen müssen. Nun kommen wir endlich zum ganz gewöhnlichen, allgemeinen Fall, daß das neue Element zwischen zwei andere gehängt werden muß:

```
  else
  // Allgemeiner Fall: Element in Liste einhängen
  { Eintrag *vor = pos->prev;
```

Die Hilfsvariable `"vor"` wird uns einige Schreibarbeit ersparen, echte Pointerartisten könnten aber auch ohne sie auskommen. Das neue Element muß jetzt zwischen die Elemente, auf die `"vor"` bzw. `"pos"` zeigen, eingehängt werden. Zuerst hängen wir es hinter `"vor"`:

```
    vor->next = neu;
    neu->prev = vor;
```

...und anschließend vor `"pos"`:

```
    neu->next = pos;
    pos->prev = neu;
}
```

Damit wären wir fertig und müssen nur noch am Ende der Funktion einen Zeiger auf das neue Element zurückgeben:

```
return neu;
}
```

Gar nicht so einfach - oder? Ich hoffe, daß ich Ihnen damit nicht ein für allemal die Lust auf dynamische Speicherorganisation genommen habe. Es gibt auch eine wesentlich einfachere Datenstruktur,

nämlich die einfach verkettete Liste, und Varianten unseres Konzepts, bei denen man gleich bei der Initialisierung der Listenstruktur „Phantom-Elemente" an den Anfang bzw. das Ende der Liste hängt, was einem die vielen Sonderfälle erspart. Aber wir bleiben unbeirrt bei unserem Listenkonzept und schreiben dafür jetzt noch zwei unverzichtbare Routinen:

2.6.6.4 Suchen und Löschen

MaxonC++ löscht zwar am Programmende automatisch alle dynamischen Objekte, aber trotzdem ist es sinnvoll, wenn man sich eine Routine schreibt, die eine ganze Datenstruktur löscht. Hier ist sie:

```
void AllesLoeschen(Liste &L)
{ Eintrag *p = L.anfang;

  while(p)
  { Eintrag *hilf = p;
    p=p->next;
    delete hilf;
  }
  Init(L);
}
```

In der Schleife wird wieder die Liste durchlaufen, wobei alle Einträge gelöscht werden. Anschließend wird der Ordnung halber `"Init"` aufgerufen, wodurch die beiden Zeiger der Listenstruktur wieder auf Null gesetzt werden.

Wieso braucht man hier den Hilfszeiger `"hilf"`? In der Schleife sind zwei Schritte auszuführen: Das Element, auf das `"p"` gerade zeigt, ist zu löschen, und `"p"` ist auf das nachfolgende Element zu setzen. Wenn man erst `"p"` weitersetzt, löscht man natürlich das falsche Element, aber wenn man erst `"delete"` aufruft, liest man anschließend bei `"p = p->next"` aus einem Datenelement, das man ja soeben gelöscht hat - eine etwas haarige Sache! Also gibt es nur eine Lösung: Man setzt erst einen Hilfszeiger auf das aktuelle Lietenelement, geht dann mit `"p"` ein Element weiter und löscht das fragliche Listenelement dann über den Hilfszeiger.

Wir wollen aber nicht immer die ganze Liste löschen, sondern auch einzelne Elemente aus der Liste entfernen. Dafür dient die folgende Funktion:

```
void Loeschen(Liste &L, Eintrag *E)
{
```

Ein Blick auf die Listenstruktur sagt uns, daß wir vor dem Löschen des Elements `"E"` dessen Vorgänger und Nachfolger verändern müssen:

```
Eintrag *vor = E->prev, *nach = E->next;
```

Es kann natürlich sein, daß `"E"` gar keinen Vorgänger hat, dann handelt es sich offenbar um den Listenanfang. Andernfalls ist der neue Nachfolger des Vorgängers der bisherige Nachfolger des zu löschenden Elements (welch ein Satz!):

```
  if (vor)
    vor->next = nach;
```

Falls wir es mit dem Listenanfang zu tun haben, müssen wir zwar nicht dessen Vorgänger (den es ja nicht gibt), wohl aber die Listenstruktur selbst verändern:

```
  else
    L.anfang = nach;
```

Dadurch wird der Nachfolger des bisher ersten Elements der neue Listenanfang. Auf der anderen Seite des zu löschenden Elements läuft alles ganz analog:

```
  if (nach)
    // Element hat Nachfolger: Dann dessen Vorgänger verändern
    nach->prev = vor;
  else
    // Kein Nachfolger, also Listenende anpassen
    L.ende = vor;
```

Nun haben wir den Eintrag aus der Liste ausgehängt und geben last not least seinen Speicher wieder frei:

```
  delete(E);
}
```

Um diese Löschfunktion zu benutzen, braucht man einen Zeiger auf ein Element, aber wie bekommt man den? Nun, zum Beispiel indem man mit der folgenden Funktion einen Eintrag aufgrund seines Namens sucht:

```
Eintrag *Suche(const Liste &L, char *Name)
{ for( Eintrag *p = L.anfang; p && strcmp(p->name, Name);
  p = p->next );
  return p;
}
```

Verglichen mit den anderen, ist diese Funktion wirklich nicht sehr kompliziert. Deshalb habe ich mir die Freiheit genommen, sie etwas konfus aufzuschreiben: Anstelle der `"while"`-Schleifen, die in den anderen Funktionen zum Durchlaufen der Liste benutzt wurde, nehmen wir hier eine `"for"`-Schleife. Vor der Schleifenausführung wird der Zeiger `"p"` mit dem Listenanfang initialisiert (wobei Sie sich nicht wundern sollten, daß `"p"` dabei deklariert wird, denn das ist in C++ erlaubt), dann folgt auch schon die Schleifenbedingung, die aus zwei Klauseln besteht. Als „echter" C-Hacker habe ich jedesmal das `"!= 0"` weggelassen, denn das ist bekanntlich zwar übersichtlicher, aber redundant. Die Schleifenbedingung könnte man also auch

```
p != 0 && strcmp(p->name, Name) != 0
```

schreiben. Der dritte Teil jeder `"for"`-Schleife ist die Anweisung, die nach jedem Durchlaufen der Schleife ausgeführt werden soll. Hier setzt sie in bekannter Weise den Zeiger `"p"` weiter.

Und was ist mit dem Schleifenrumpf? Nun, wir tun doch schon alles, was nötig ist, deshalb entfällt er hier ersatzlos, und er wird durch ein schlichtes, einsames Semikolon ersetzt. Am Ende der Funktion

müssen wir nur noch den Wert von `"p"` zurückgeben, denn dieser zeigt dann entweder auf die gesuchte Person oder ist Null, wenn der Name nicht gefunden wurde. Dies ist wieder ein nettes Beispiel dafür, daß man in C ausgesprochen kompakte Programme schreiben kann, die dann nicht gerade gut lesbar sind.

2.6.6.5 Das Programm an einem Stück

Aus Gründen der besseren Übersicht folgt jetzt noch einmal das Ganze Listenprogramm. Außerdem braucht jedes Programm ein Hauptprogramm, weshalb ich noch eine kleine Funktion `"main"` angehängt habe.

```
/*
Doppelt verkettete Listen

Ein kleines Beispiel für dynamische Datenstrukturen in C++, aber ohne
objektorientierte Programmierung

Programmiert von Jens Gelhar am 13.01.92
*/

#include <stream.h>
#include <string.h>

// * * * Datentypen * * *

struct Eintrag        // Ein einzelner "Telefonbuch"-Eintrag
{ Eintrag *next, *prev;   // Vorgänger und Nachfolger

  char name[30];        // Name und...
  char nummer[20];    // Telefonnummer
};

struct Liste          // Sortierte Liste von Einträgen
{ Eintrag *anfang, *ende;    // Erstes bzw. letztes Listenelement
};

// * * * Funktionen * * *

void Init(Liste &L)   // Initialisiert eine Listenstruktur,
                              // indem sie ihre beiden
                              // Pointer auf 0 setzt:
{ L.anfang = 0;
  L.ende   = 0;
}

void Ausgabe (const Liste &L)  // Telefonbuch ausgeben
{ for(Eintrag *E = L.anfang; E; E = E->next)
    cout « E->name « ": " « E->nummer « "\n";
}

Eintrag *Einfueg(Liste &L, const char Name[], const char Nummer[])
  // Fügt einen neuen Eintrag in die Liste ein
```

```
{ // Neues Element erzeugen:
  Eintrag *neu = new Eintrag;

  if (!neu) return 0;        // Kein Speicher frei!

  // Neues Element initialisieren:
  strcpy(neu->name, Name);
  strcpy(neu->nummer, Nummer);

  // Richtige Position zun Einfügen suchen:
  Eintrag *pos = L.anfang;

  while (pos!=0 && strcmp(pos->name, Name) < 0)
    pos = pos->next;

  // Element in Liste einhängen, dabei diverse
  // Sonderfälle beachten:

  if (pos == 0) // Sonderfall: Anfügen an das Ende der Liste
  { Eintrag *alt = L.ende;  // Altes Listenende

    if (alt != 0) // Allgemeiner Unter-Fall: wirklich anhängen
    { alt->next = neu;
      neu->prev = alt;
      neu->next = 0;
      L.ende = neu;
    }
    else        // Spezieller Spezialfall: Liste ist noch leer
    { neu->next = 0;
      neu->prev = 0;
      L.anfang = neu;
      L.ende = neu;
    }
  }
  else
    if (pos->prev == 0)   // Sonderfall: Neues Element an
                                        // Listenanfang hängen
    { pos->prev = neu;
      neu->next = pos;
      neu->prev = 0;
      L.anfang = neu;
    }
    else
    // Allgemeiner Fall: Element in Liste einhängen
    { Eintrag *vor = pos->prev;
      // "neu" zwischen "vor" und "pos" einhängen:
      vor->next = neu;
      neu->prev = vor;
      neu->next = pos;
      pos->prev = neu;
    }
```

```
  // Fertig, Zeiger auf neues Element zurückgeben:
  return neu;

}

void AllesLoeschen(Liste &L)
// Löscht sämtliche Listenelemente
{ Eintrag *p = L.anfang;

  while(p)
  { Eintrag *hilf = p;
    p=p->next;
    delete hilf;
  }
  Init(L);
}

void Loeschen(Liste &L, Eintrag *E)
  // Löscht das Element, auf das "E" zeigt, aus Liste "L"
{ // Vorgänger unf Nachfolger des zu löschenden Elements
  Eintrag *vor = E->prev, *nach = E->next;

  if (vor)  // Element hat Vorgänger: dann dessen
              // Nachfolger umsetzen
    vor->next = nach;
  else      // Kein Vorgänger: dann Listenanfang aktualisieren
    L.anfang = nach;

  if (nach) // Element hat Nachfolger: dann dessen Vorgänger
              // verändern
    nach->prev = vor;
  else      // Kein Nachfolger, also Listenende anpassen
    L.ende = vor;

  delete(E);
}

Eintrag *Suche(const Liste &L, char *Name)
// Sucht einen Eintrag mit diesem Namen und
// gibt Zeiger auf das
// zugehörige Listenelement zurück, oder 0 bei Fehler
{ for( Eintrag *p = L.anfang; p && strcmp(p->name, Name);
    p = p->next );
  return p;
}

// * * *  Hauptprogram  * * *

void main()
{ // Eine Liste wird deklariert und initialisiert:

  Liste l;
```

```
  Init(l);

  // Ein paar Beispieldaten werden eingefügt:

  Einfueg(l, "Harley-Davidson", "001-414/342-4680");
  Einfueg(l, "Boris Becker", "0815/4711");
  Einfueg(l, "James Bond", "007/26731");
  Einfueg(l, "Maxon", "06196/481811");
  Einfueg(l, "Zaphod Beeblebrox", "00042/08154242");
  Einfueg(l, "Luxemburg", "00352");
  Einfueg(l, "Queensr\0377che", "795069");
  Einfueg(l, "Fishbone", "4676152");

  // Einen Datensatz suchen und ggf. löschen:

  Eintrag *e = Suche(l,"Maxon");
  if (e)
    Loeschen(l, e);
  else
    cout « "Name nicht gefunden!\n";

  // Restliche Liste ausgeben:
  Ausgabe(l);

  // ...und wieder aufräumen:
  AllesLoeschen(l);

}
```

3. Objektorientiertes Programmieren

3.1 Klassen und Objekte

3.1.1 Keine Panik!

Viele Leute behaupten, das Konzept der objektorientierten Programmierung werde die Softwareentwicklung revolutionieren, weit stärker als das etwa die Einführung der strukturierten Programmiersprachen es in den beiden vergangenen Jahrzehnten getan hat, und wahrscheinlich werden diese Leute recht behalten.

Die Sache hat bisher einen kleinen Haken: So mancher, der von „objektorienterter Programmierung" spricht, hat keine rechte Ahnung davon, was das eigentlich ist. Auch viele erfahrene Programmierer stehen ratlos vor Begriffen wie „Klasse", „Methode" oder „Vererbung" und verstehen die Welt nicht mehr. Was ist geschehen? Wird jetzt alles anders? Sind fünf Jahrzehnte prozeduraler Programmierung plötzlich nichts mehr wert?

Aber die Rettung naht, ist sogar schon da: Kapitel 3 des Handbuchs zu MaxonC^{++}, jawohl, genau das, was Sie jetzt gerade lesen!

Zunächst sollten Sie das bekannte Motto, das in großen, freundlichen Buchstaben über diesem Abschnitt steht, beherzigen. Damit das Ihnen etwas leichter fällt, werde ich im folgenden versuchen, halbwegs klarzustellen, was es mit der objektorientierten Programmierung eigentlich auf sich hat.

Das Problem dabei, das auch zu der allgemeinen Verwirrung und Unkenntnis im Zusammenhang mit OOP („objekt-orientierter Programmierung") erheblich beiträgt, ist, daß das Ganze nur bei großen, komplexen Projekten Sinn macht. Es ist zwar durchaus möglich, ein objektorientiertes „Hello World"-Programm zu schreiben, aber einen sittlichen Nährwert hat das nicht. Aber nehmen wir als Beispiel doch ganz einfach das Listen-Programm, das am Ende von Kapitel zwei ausführlich dargestellt wurde. Darin könnte man nämlich schon einige (mit etwas Mühe sogar alle) objektorientierten Features sinnvoll einsetzen.

Eine Grundidee der objektorientierten Programmierung ist, den Daten des Programms eine gewisse Intelligenz zu verleihen. Bei größeren Programmen stellt man schnell fest, daß man eigentlich herzlich wenig mit reinen Algorithmen zu tun hat, im wesentlichen bestehen viele Programme aus Datenstrukturen und Funktionen, die auf diesen Daten operieren. Oft liegt die ganze Cleverness beim Programmieren im geschickten Design der Datenstrukturen, während die Algorithmen des Programms vergleichsweise trivial sind. Und damit wären wir auch schon bei einigen im Zusammenhang mit OOP vielbenutzten Begriffen:

Eine „Klasse" ist zunächst nichts anderes als ein ganz gewöhnlicher Datentyp, wie Sie ihn aus der „konventionellen" (d. h. nicht objektorientierten) Programmierung kennen, nur daß zu einer Klasse in der OOP zusätzlich noch Funktionen gehören können. Die Daten und die darauf definierten Funktionen einer Klasse nennt man zusammen sehr vornehm auch die „Eigenschaften" der Klasse.

In C++ ist das grundlegende Sprachkonstrukt zur Definition von Klassen die Struktur, die es ja auch schon in C gibt. Nun kann man in C++ als Member einer Struktur nicht nur Daten, sondern auch Funktionen deklarieren. Diese Funktionen werden dadurch formal an die Struktur, oder sagen wir lieber Klasse, angebunden. Zum Beispiel haben wir im Telefonlisten-Programm zuerst den Datentyp `"Liste"` und anschließend eine Funktion namens `"Einfueg"` definiert, die dann ziemlich beziehungslos im Programmtext 'rumlagen. In C++ kann man die Funktion `"Einfueg"` direkt in die Klassendefinition hineinschreiben, so daß sie Bestandteil der Klasse und ihrer Eigenschaften wird. Solche Funktionen bezeichnet man auch als „Methoden", aber ich persönlich mag diesen Begriff nicht und ziehe die Bezeichnung „Member-Funktion" vor, zumal auch Stroustrup die schlicht Bezeichnung „Member function" bevorzugt.

Man erhält nun eine klare Trennung zwischen den Funktionen, die zu einer Klasse gehören, und allen anderen, die nur „von außen" auf die Klasse zugreifen und die Klasse und ihre Eigenschaften lediglich benutzen. Der Gedanke liegt nahe, den beiden Funktionstypen unterschiedliche Rechte einzuräumen. In C++ dürfen die Member-Funktionen stets „alles" mit einem Klassenobjekt tun, während der Programmierer festlegen kann, welche Eigenschaften der Klasse „der Allgemeinheit" zur Verfügung stehen. Auch das läßt sich wieder an unserer Klasse `"Liste"` veranschaulichen: Hier bietet es sich an, die beiden Datenmember `"anfang"` und `"ende"` als `"privat"` zu deklarieren, so daß man diese von außen nur indirekt über Funktionen wie `"Init"`, `"Einfueg"` usw. benutzen kann. Dadurch ist sichergestellt, daß kein Programmteil versehentlich an diesen beiden Zeigern herumspielen kann. Auch bei der Klasse bzw. Struktur `"Eintrag"` würde man am besten die Member `"prev"` und `"next"` als privat deklarieren, denn diese sind für den „Benutzer" der Klasse nicht von Interesse, während man `"name"` und `"nummer"` eher als öffentlich deklarieren würde. C++ bietet einige durchdachte Features, um derartiges „Information Hiding" zu realisieren und so zwischen den internen Daten und der öffentlichen „Schnittstelle" einer Klasse zu unterscheiden, was als „Separation of Concerns" („Trennung von Angelegenheiten") bezeichnet wird.

Im Telefonlisten-Programm wird Ihnen vielleicht aufgefallen sein, daß man ein Objekt des Typs `"Liste"` erst mit der Funktion `"Init"` initialisieren muß, bevor man es benutzen kann, und daß man die Funktion `"AllesLoeschen"` aufrufen muß, wenn man die Liste nicht mehr benötigt. In C++ und anderen OO-Programmiersprachen stehen Ihnen Features zur Verfügung, die derartige Vorgänge automatisieren, nämlich **„Konstruktoren"** und **„Destruktoren"**. Man deklariert einfach einen Konstuktor, der ein `"Liste"`-Objekt bei seiner Erzeugung initialisiert, und einen Destruktor, der aufräumt, sobald die Existenz des Objekts endet, und schon weiß der Compiler, was er zu tun hat. Das spart nicht nur viel Arbeit, sondern sichert auch die Konsistenz. Man kann darauf vertrauen, daß eine `"struct Liste"` stets sinnvolle Dateneinträge hat. Dadurch beginnen Datenobjekte tatsächlich, ein gewisses Eigenleben zu führen und so etwas wie eine „ lokale Intelligenz" zu besitzen.

Das populärste Schlagwort der OOP-Fans ist sicherlich die „Vererbung". Das klingt furchtbar sophisticated, ist aber letzten Endes doch sehr trivial.

Um wieder auf das kanonische Beispiel zurückzukommen: Ganz unabhängig davon, ob man in einer Listenstruktur nun Telefonnummern oder spanische Inquisitoren abspeichert, gibt es Probleme und

Lösungen, die immer wieder vorkommen. Wenn man ein Datenelement in die Liste einfügt, hängt es durchaus von diesen Daten ab, WO das neue Element eingehängt werden soll, aber das WIE, nämlich dieses furchtbar komplizierte Umhängen von Pointern, ist immer dasselbe. Auch die Löschfunktion könnte man für alle möglichen Listen verwenden.

Der objektorientierte Ansatz ermöglicht es, zunächst ganz allgemein eine Klasse `"Listeneintrag"` zu definieren und dann davon eine neue Klasse, etwa `"Telefonbucheintrag"`, abzuleiten. Die abgeleitete Klasse „erbt" dann alle Eigenschaften der „Basisklasse" und kann neue hinzufügen, nötigenfalls auch Eigenschaften der Basisklasse in kontrollierter Weise verändern (Stichwort: „virtuelle Member"). Und das ist auch fast schon alles, was es mit der objektorientierten Programmierung, jedenfalls der Art von OOP, die C++ zu bieten hat, auf sich hat, wobei übrigens die oben gewählte Reihenfolge der Konzepte nicht unbedingt mit deren Wichtigkeit zu tun hat.

Ihnen brummt der Schädel? Dann wird es höchste Zeit, daß es konkret wird, und nicht vergessen: Keine Panik!

3.1.2 Member-Funktionen

3.1.2.1 Erste Beispiele

Strukturen sind relativ unhandliche Gebilde, die zudem von Haus aus ziemlich wenig können. Zum Beispiel kann man sie nicht so ohne weiteres über `"cout"` ausgeben, und so kommt es, daß man oft eine Ausgabefunktion zu einer Struktur schreiben muß. Dies soll als erstes Beispiel für eine Member-Funktion dienen:

```
struct Person
{ char Vorname[20], Nachname[20];
  int Tag, Monat, Jahr;
  void Ausgabe();
};

Person p1, *person_ptr;
```

Urplötzlich und unerwartet taucht hier in der Strukturdefinition ein Funktionsprototyp auf. Dadurch wird eine Funktion namens `"Ausgabe"` als Member (oder „Methode") der Klasse `"Person"` deklariert. Nun kann man sie für jedes Datenobjekt aufrufen:

```
p1.Ausgabe();
person_ptr->Ausgabe();
```

Dafür ist es natürlich wie immer nötig, die Funktion zu definieren. Dabei muß man dem Compiler aber irgendwie klarmachen, daß man die Member-Funktion `"Ausgabe"` der Struktur `"Person"` zu definieren gedenkt und nicht irgendeine andere Funktion gleichen Namens, denn genau wie alle anderen Member liegt auch `"Ausgabe"` hier im Gültigkeitsbereich der Struktur verborgen. Dazu dient der Scope-Operator `"::"`. Mit `"Person::Ausgabe"` kann man von jeder Stelle des Programms aus genau diese Memberfunktion ansprechen, und das benutzen wir auch, wenn wir sie definieren:

```
void Person::Ausgabe()
{ // usw.
```

Scheinbar hat die Funktion keine Parameter, scheinbar, denn eine normale Memberfunktion hat immer einen Parameter namens `"this"`. Das ist ein Zeiger auf das Objekt, auf dem die Funktion aufgerufen wird. Also können wir folgendes schreiben:

```
void Person::Ausgabe()
{ cout « this->Nachname « ", " « this->Vorname « "\n";
  cout « "geboren am " « this->Tag « "." « this->Monat « "."
  « this->Jahr « "\n";
}
```

Das könnte man schreiben, muß man aber nicht! Die erste Deklaration einer Member-Funktion liegt im Scope ihrer Klasse, und so kommt es, daß auch bei der späteren Funktionsdefinition jener Scope sichtbar ist. Also sind die Membernamen wie `"Vorname"` oder `"Monat"` in der Funktion `"Person::Ausgabe"` direkt sichtbar und bezeichnen dort die entsprechenden Member von `"*this"`. Die folgende Funktion ist viel kürzer und schöner:

```
void Person::Ausgabe()
{ cout « Nachname « ", " « Vorname « "\n";
  cout « "geboren am " « Tag « "." « Monat « "." « Jahr « "\n";
}
```

Auch Member-Funktionen können auf diese einfache Weise aufgerufen werden:

```
struct S
{ int i;
  void f(int);
  int g(int);
};

int S::g(int j)
{ f(j);          // statt "this->f(j);"
  return i;   // statt "this->i;"
}
```

Eine Member-Funktion kann prinzipiell einen Parameter ihres Klassentyps besitzen:

```
struct T1
{ void memberfunktion(T1);
};
```

Beim Aufruf einer solchen Funktion muß aber die ganze Klasse auf den Stack kopiert werden, was bisweilen zu fürchterlich langsamen Programmen führt. Deshalb ist es in der Regel sinnvoller, wenn man als Parameter eine Referenz auf ein konstantes Objekt deklariert:

```
struct T2
{ void memberfunktion(const T2&);  // Meist besser
};
```

Andererseits kann eine Member-Funktion ein Objekt ihres Klassentyps als Ergebnis zurückgeben:

```
struct T3
{ T3 memfunkt();// OK
};
```

3.1.2.2 Member-Funktionen (Methoden) und konstante Objekte

Da wir gerade beim Thema "const" sind: "this" zeigt ja normalerweise auf ein unqualifiziertes Klassenobjekt, was natürlich Probleme bereitet, wenn man eine solche Funktion auf einem als "const" qualifizierten Objekt aufrufen würde, denn die Memberfunktion könnte das Objekt über "this" beliebig verändern. Deshalb kann man den Typ, auf den "this" zeigt, auf etwas unkonventionelle Weise verändern, indem man nämlich "const" oder "voliatile" an die Funktionsdeklaration anhängt:

```
struct Quali
{ int Member;
  void normal();
  void konstant() const;  // d. h. Objekt wird nicht verändert
};

void Quali::normal()
{ Member = 42;  }       // OK

void Quali::konstant() const
{ Member = 26731;  } // ERROR, "Member" ist hier "const"

void main()
{ Quali q1;
  const Quali q2;

  q1.normal();     // OK
  q1.konstant();// OK

  q2.normal();     // ERROR, "q2" ist "const", "normal" aber nicht!
  q2.konstant();// OK   }
```

Allgemein empfiehlt es sich, mit "const"-Qualifizierungen nicht zu geizen und jede Member-Funktion, die ihr Objekt nicht verändert, als "const" zu deklarieren.

3.1.2.3 Inlining von Member-Funktionen

Member-Funktionen können als "inline" deklariert werden:

```
struct Klasse1
{ int i;
  inline void set_i(int);
  inline int get_i();
};

void Klasse1::set_i(int j)   // "inline" muß man nicht nochmal
{ i = j; }                          // setzen, darf man aber!
```

```
int Klasse1::get_i()
{ return i; }
```

Das ist natürlich eine ganze Menge Schreibaufwand für zwei triviale Minifunktionen, weshalb man das auch abkürzen kann: Wenn die Funktionsdefinition direkt in die Klasse hineingeschrieben wird, gilt die Funktion automatisch als `"inline"`:

```
struct Klasse1
{ void set_i(int j)  // Automatisch "inline"
  { i = j;
  }
  int get_i()          // Das auch
  { return i;
  }

  int j;
};
```

Solche `"inline"`-Definitionen werden zuerst nur syntaktisch analysiert und erst am Ende der Klassendefinition auf Semantik getestet und wirklich definiert. Deshalb können solche Funktionen scheinbar auf Namen zugreifen, die noch nicht deklariert sind, wie es z. B. oben mit `"i"` geschehen ist.

Sie denken, das Beispiel oben sei ziemlich konstruiert und sinnlos? Denkste, so etwas benutzt man tatsächlich des öfteren, dann nämlich, wenn ein Member `"privat"` ist und man „von außen" aus Sicherheitsgründen nur kontrolliert durch derartige Funktionen darauf zugreifen können soll, was wie immer eine elegante Überleitung zum folgenden Abschnitt darstellt:

3.1.3 Klassen und Zugriffsrechte

Wie zuvor bereits wiederholt angedeutet, unterstützt C++ das Konzept des „Information Hiding" durch unterschiedliche Zugriffsrechte auf Class-Member. Dafür gibt es zunächst die beiden Schlüsselwörter `"private"` und `"public"`, z. B.

```
struct Namensliste
{ private:
    Namensliste *link;
  public:
    void Einfueg (Namensliste &vorgaenger);
    char Vorname[20], Nachname[20];
};

void Namensliste::Einfueg (Namensliste &vorgaenger)
{ link = vorgaenger.link;
  vorgaenger.link = this;
}

Namensliste nl;
```

```
#include <stream.h>

void main()
{ nl.link = 0;              // ERROR
  cout « nl.Vorname; // OK
}
```

Die Memberfunktion `"Einfueg"` darf auf alle Member der Struktur zugreifen, einschlielich des privaten Members `"link"`, der so etwas wie einen Verbindungszeiger einer einfach verketteten Liste darstellen soll. Wenn dagegen die Funktion `"main"`, die kein Member der Klasse `"Namensliste"` ist, an `"link"` rumfummeln will, kloppt ihr der Compiler auf die Pfoten.

Bei diesem Konzept der Zugriffskontrolle sind zwei Dinge bemerkenswert:

- Es handelt sich wirklich um eine Prüfung auf Rechte und nicht um eine variable Sichtbarkeit. Bei `"nl.link"` in `"main"` ist der Bezeichner durchaus sichtbar, und der Compiler weiß, was gemeint ist, so daß MaxonC++ hier durchaus den entsprechenden Code erzeugt. Es wird lediglich gewarnt, daß `"main"` auf diesen Member nicht zugreifen darf.
- Die Zugriffskontrolle bezieht sich grundsätzlich auf Bezeichner, nicht etwa auf Objekte:

```
struct Beispiel
{ private:
    int i;
  public:
    void rumfummeln(Beispiel*);
};

Beispiel b1, b2;

void Beispiel::rumfummeln(Beispiel *bp)
{ bp->i = 26731;
}

void main()
{ b1.rumfummeln(&b2);
}
```

Im Beispiel wird die Member-Funktion `"rumfummeln"` auf `"b1"` mit einem Zeiger auf `"b2"` als Argument aufgerufen, so daß `"this"` auf `"b1"`, der Parameter `"bp"` aber auf ein ganz anderes Objekt zeigt. Trotzdem darf `"rumfummeln"` an dem privaten Member `"i"` dieses völlig anderen Objekts 'rumfummeln, und ebenso dürfen alle Objekte einer Klasse unbeschränkt auf alle anderen Objekten derselben Klasse zugreifen.

Es ist natürlich egal, in welcher Reihenfolge man die privaten und die öffentlichen Member einer Klasse deklariert. Nach dem Schlüsselwort `"struct"` ist der Zugriffsmodus als Default `"public"` - das muß ja auch so sein, denn sonst wäre C++ ja nicht mehr zu C kompatibel. Aber es gibt daneben

auch noch das Schlüsselwort `"class"`, das (fast) identisch mit `"struct"` ist, mit einem Unterschied: bei `"class"` ist `"private"` der voreingestellte Zugriffsmodus, z. B.

```
class Beispiel
{    int i;       // Ist jetzt automatisch "private"
  public:
    void rumfummeln(Beispiel*);
};
```

Ansonsten ist eine `"class"` absolut identisch mit einer `"struct"`. Im folgenden werde ich hauptsächlich `"class"` verwenden, ganz einfach deshalb, weil das so schön objektorientiert klingt und dadurch total hip ist.

Bei überladenen Funktionsnamen dürfen die einzelnen Funktionen durchaus unterschiedliche Zugriffsrechte besitzen:

```
class C
{    void f(int);
  public:
    void f(double);
} c1;

void main()
{ c1.f(1);       // ERROR
  c1.f(1.0);     // OK
}
```

Hier sieht man noch einmal, daß die Zugriffskontrolle in C++ nicht über Sichtbarkeit, sondern über eine tatsächliche Rechte-Prüfung geschieht. Die Funktion `"f(int)"` ist in `"main"` sichtbar, so daß der Compiler beim ersten Funktionsaufruf auch erkennt, daß diese Funktion gemeint ist. Wäre sie nicht sichtbar, würde er das Argument `"1"` einfach nach `"double"` wandeln und die gerade einzig sichtbare, öffentliche Funktion aufrufen.

Übrigens gibt es noch einen dritten Zugriffsmodus namens `"protected"`, der aber nur im Zusammenhang mit Vererbung nützlich und sonst mit `"private"` identisch ist. `"protected"` wird in 3.2.2 näher beschrieben.

3.1.4 No Scope, no hope!

Es ist leider wieder einmal an der Zeit, etwas zur Gültigkeit von Namen zu sagen. Wie Sie sicher schon mitbekommen haben, hat jede Klasse ihren eigenen Scope (Gültigkeitsbereich). In C++ ist das allerdings ein „vollständiger" Scope, während in C nur die Bezeichner der Member darin lagen. Dieser Sachverhalt hat einige unangenehme Konsequenzen, z. B. das kanonische Beispiel, das schon an anderer Stelle dieses Manuals zitiert wurde:

```
class X
{ class Y *ypern;    // 1914
};
```

```
class Y
{ int Schmonz;        // ...
}
```

Laut Standard-C++ sind die Klasse "Y", die in "X" referiert wird, und die globale Klasse "Y" absolut verschieden, aber ich erwähnte bereits, daß MaxonC++ solche Fälle beherrscht. Andererseits hat diese Scope-Regelung auch positive Seiten: Man kann "typedef"-Namen und "enum"-Typen deklarieren, die ausschließlich innerhalb einer Klasse sichtbar sind:

```
class Geheim
{ public:
     typedef char *strptr;
     strptr Vorname, Nachname;
     void f();
};

strptr NochEinName;   // ERROR

void Geheim::f()
{ strptr x1;              // OK
                               // usw...
}
```

Der Bezeichner "strptr" liegt im Scope der Klasse "Geheim" und ist deshalb auf Dateiebene nicht sichtbar, wohl aber in der Definition der Member-Funktion "f". Man kann solche Bezeichner aber von außen sichtbar machen, und zwar mit dem Scope-Operator "::", den wir dazu auch schon bei Definitionen von Member-Funktionen benutzen:

```
class Geheim
{ public:
     typedef char *strptr;
     // usw.
};

Geheim::strptr UndNochEinName;    // So geht's!
```

Natürlich gelten hier wieder die bekannten Zugriffsregeln: Wäre "strptr" als privat deklariert gewesen, würde der Compiler wieder eine entsprechende Warnung ausgeben.

Wie bei allen ineinander verschachtelten Scopes, können auch in einer Klasse Bezeichner höherer Ebenen überdeckt werden. Dann kann man mit "::" ohne vorangestellten Klassennamen gezielt Bezeichner der Dateiebene ansprechen:

```
int i;

class S
{ public:
     int i;
     void f();
};
```

```
void S::f()
{ i = 1;          // Entspricht "this->i"
  S::i = 2;    // Das auch
  ::i = 3;     // Dies ist das globale "i"
}
```

Einen Namen mit einem `"::"` bezeichnet man auch als „qualifiziert", was aber nichts mit den Typ-Qualifikationen `"const"` und `"volatile"` zu tun hat.

3.1.5 Friends

Das mit den unterschiedlichen Zugriffsrechten ist zwar schön und gut, kann einem aber auch ganz gewaltig auf die Nerven gehen. Betrachten wir wieder einmal das Telefonlisten-Programm. Hier ist es sicher sinnvoll, wenn die beiden Verbindungszeiger der Klasse `"Eintrag"` privat sind, denn den „Benutzer" dieser Listen soll es möglichst nicht interessieren, wie die Listenstruktur implementiert ist. Andererseits wird man Operationen wie `"Einfuegen"` wohl als Member-Funktionen der Klasse `"Liste"` definieren, und die müssen ja normalerweise auf die Link-Zeiger der Listeneinträge zugreifen. Es fehlt also bisher ein Sprachkonstrukt, mit dem man so etwas deklarieren kann wie „Diese Bezeichner sind privat, aber die Funktion `xxx` darf sie trotzdem benutzen", und genau dazu dienen `"friend"`-Deklarationen:

```
class C
{ int member;       // Privat!
  friend void f();
};

C c1;

void f()
{ c1.member++;      // OK
}

void g()
{ c1.member++;      // Fehler
}
```

In der Klasse `"C"` wird die Funktion `"f"` als „Freund" der Klasse deklariert, weshalb sie Zugriff auf alle privaten Member von `"C"` besitzt. `"g"` ist kein `"friend"`, also bemäkelt der Compiler hier wie gewohnt einen Zugriffsfehler.

Natürlich kann eine Funktion `"friend"` beliebig vieler Klassen sein. Sie ist nicht Bestandteil der Klasse (wie eine Member-Funktion) und zählt eigentlich auch nicht zu den „Eigenschaften" der Klasse, sondern hat umgekehrt selbst eine Eigenschaft, nämlich die vollen Rechte an der Klasse, deren Freund sie ist. In ihr ist auch der Scope jener Klasse nicht unmittelbar sichtbar.

Bei überladenen Funktionen bezieht sich die `"friend"`-Eigenschaft ausschließlich auf die Instanz, die als `"friend"` deklariert wurde:

```
class S
{ int member;
  friend void f();
  friend void f(int);
} s1;

void f()
{ s1.member = 1;
}           // OK

void f(int i)
{ s1.member = i;
}           // OK

void f(char c)
{ s1.member = c;
}           // ERROR, kein "friend" von "S"
```

Interessanterweise werden an dieser Stelle die sonst so strikten Scope-Regeln durchbrochen: Die Funktion `"f"` wird bei der `"friend"`-Deklaration, also innerhalb der Klassendefinition, zum ersten Mal erwähnt, gehört aber trotzdem in den äußeren Scope.

Wenn man einer Member-Funktion einer anderen Klasse die `"friend"`-Eigenschaft zugesteht, wird das seltsamerweise nicht so locker wie bei globalen Funktionen gehandhabt:

```
class T;

class S
{ friend void T::f();      // ERROR: "f" ist unbekannt
};

class T
{ void f();
};
```

In diesem Fall muß die `"friend"`-Funktion tatsächlich schon in ihrem eigenen Scope deklariert worden sein, etwa so:

```
class T
{ void f();
};

class S
{ friend void T::f();      // In dieser Reihenfolge OK
};
```

Das führt zu unangenehmen Situationen, wenn zwei Klassen ihren Funktionen gegenseitig den `"friend"`-Status gönnen wollen:

```
class T;

class S
```

```
{ void sfun();
  friend void T::tfun();     // Error
};

class T
{ void tfun();
  friend void S::sfun();
};
```

Was tun? Einfaches Vertauschen der Klassendefinitionen hilft uns hier offensichtlich nicht aus der Klemme, sondern nur ein neues Feature: Man darf auch eine ganzen Klasse einschließlich aller ihrer Member-Funktionen als Freund deklarieren:

```
class S
{ void sfun();                  // Privat!
  typedef char *strptr; // Privat!
  friend class T;             // Ganze Klasse "T" ist Freund
};

class T
{ S::strptr tmember;       // OK, "T" ist friend von "S"
  void tfun();
};

void T::tfun()
{ S s1;
  s1.sfun();                    // OK, Freund von "S"
}
```

Man beachte dabei, daß Freundschaft in C++ im allgemeinen eine recht einseitige Angelegenheit ist: Im Beispiel ist `"T"` ein Freund von `"S"`, aber nicht umgekehrt.

3.1.6 Statische Member

Alles, was mit einer Klasse zu tun hat und ausschließlich von den Member-Funktionen der Klasse benutzt wird, sollte man auch als Member deklarieren und so im Scope der Klasse verstecken. Dabei kommt es allerdings vor, daß ein Objekt oder eine Funktion nicht sinnvoll an ein Objekt gebunden werden kann, sondern eigentlich eher global ist. Um das Gesülze konkret zu machen: Die Klasse `"Person"`, die einen Namen nebst Adresse enthalten soll, hat auch eine Funktion namens „Drucken", die die Daten auf den Drucker ausgeben soll. Dazu braucht sie aber einen Filedeskriptor, der (aus welchen Gründen auch immer) ausschließlich in dieser Klasse benutzt wird und folglich auch dort hinein gehört. Es wäre aber nicht sinnvoll, in jeden einzelnen Datensatz diesen Deskriptor aufzunehmen, denn das wäre ja Speicherverschwendung.

Außerdem braucht man wenigstens noch eine Funktion, die die Druckerdatei öffnet, was natürlich nicht von einem konkreten Objekt abhängig ist und folglich keinen `"this"`-Zeiger erfordert.

Normalerweise würde man das so implementieren:

```
#include <stdio.h>              // Für Dateien und so'n Zeugs

class Person
{ public:
     char Name[40], Adresse[50];
     void Drucken();
};

FILE *DruckerDatei;             // Filedeskriptor für Drucker

void DruckerOeffnen()
{ DruckerDatei = fopen("prt:", "w"); // Macht Drucker klar
}

void Person::Drucken()
{ fprintf(DruckerDatei, "%s\n", Name);
  fprintf(DruckerDatei, "%s\n", Adresse);
}
```

Mit dem Schlüsselwort `"static"` kann man nun sowohl die Variable `"DruckerDatei"` als auch die Funktion `"DruckerOeffnen"` in den Scope von `"Person"` verlegen, ohne daß sie an ein Objekt gebunden wären:

```
#include <stdio.h>              // Für Dateien und so'n Zeugs

class Person
{ public:
     static FILE *DruckerDatei;
     static void DruckerOeffnen();
     char Name[40], Adresse[50];
     void Drucken();
};

FILE* Person::DruckerDatei;

void Person::DruckerOeffnen()
{ DruckerDatei = fopen("prt:", "w"); // Macht Drucker klar
}

void Person::Drucken()
{ // genau wie oben ...
}
```

Auch wenn die beiden Member mit dem Schlüsselwort `"static"` deklariert werden, haben sie trotzdem externe Linkage, können also von anderen Modulen importiert und benutzt werden. Wir haben es also mit einer weiteren Bedeutung dieses ohnehin schon überladenen Wortsymbols zu tun. Bei einem Datenmember wird sogar `"extern"` als Speicherklasse angenommen, d. h. der Compiler geht zunächst davon aus, daß diese Variable von woanders importiert werden muß. Erst durch die erneute Deklaration von `"Person::DruckerDatei"` außerhalb der Klassendefinition wird oben ein entsprechendes Datenobjekt erzeugt.

Der Zugriff auf einen statischen Member erfolgt innerhalb des Scopes der Klasse (also insbesondere aus ihren Member-Funktionen heraus) wie gewohnt, also einfach durch ihren Namen. Von „außerhalb" (und natürlich auch von innerhalb, wenn man das unbedingt will) hat man die Auswahl: Erstens darf man sie wie normale, nicht-statische Member über ein Objekt der Klasse benutzen:

```
Person P1;

void main()
{ P1.DruckerDatei = 0;
    P1.DruckerOeffnen();
}
```

Zweitens sind statische Member bekanntlich nicht an ein bestimmtes Objekt gebunden, weshalb man sie auch über qualifizierte Bezeichner aufrufen kann:

```
void main()
{ Person::DruckerDatei = 0;
    Person::DruckerOeffnen();
}
```

Es versteht sich von selbst, daß auch hier die üblichen Zugriffskontrollen durchgeführt werden, d. h. `"main"` darf private statische Member von `"Person"` nicht benutzen.

3.2 Vererbungslehre

3.2.1 Abgeleitete Klassen

3.2.1.1 Ein erstes Beispiel

Das Konzept der Vererbung ist das zentrale Element der objektorientierten Programmierung, und es ist an der Zeit, es hier endlich vorzustellen.

Eine Klasse von Objekten, wie Sie sie bisher kennengelernt haben, repräsentiert die Eigenschaften eines bestimmten Konzepts, einer Idee, vielleicht auch eines Objekts der realen Welt. Dabei macht man jedoch schnell die Beobachtung, daß unterschiedliche Klassen zum Teil identische Eigenschaften haben, daß es oft so etwas wie eine Klassenhierarchie gibt. Ein vielzitiertes Beispiel: Die Klasse `"Angestellter"` soll die Beschäftigten einer Firma implementieren, etwa so:

```
class Angestellter
{ private:
    // Irgendwelche implementatorische Details
  public:
    char Vorname[20], Nachname[20];
    unsigned int Gehalt;
    // usw..
};
```

Nun gibt es natürlich unterschiedliche Arten von Angestellten; in unserer Firma interessieren uns vor allem Entwickler, Verkäufer und Manager, wobei die Entwickler sich wiederum in Programmie-

rer und Hardwareentwickler unterteilen (Leute für die Produktion brauchen wir nicht, denn wir lassen sowieso in Taiwan produzieren, und der Pförtner hat normalerweise keine besonders aufregenden Eigenschaften und ist deshalb ein ganz normaler Angestellter, wie ihn die gleichnamige Klasse definiert). Jeder Entwickler sollte an einem Projekt arbeiten, deshalb braucht die Klasse `"Entwickler"` einen zusätzlichen Eintrag für den Projektnamen, während sie sonst alle anderen Eigenschaften der Klasse `"Angestellter"` übernehmen („erben") soll. Das geht in C++ ganz einfach:

```
class Entwickler : public Angestellter
{ public:
     char Projekt[25];
};
```

Die Klassendefinition beginnt mit einem Doppelpunkt `":"` und dem Namen der Klasse `"Angestellter"`, was ganz einfach besagt, daß die neue Klasse `"Entwickler"` alle Eigenschaften der Klasse `"Angestellter"` erben soll. Man bezeichnet `"Angestellter"` als `"Basisklasse"` von `"Entwickler"`, während umgekehrt `"Entwickler"` eine „abgeleitete Klasse" ist. In anderen Programmiersprachen nennt man den gleichen Sachverhalt „Subklasse" und „Superklasse", was sich aber als nicht sehr glücklich erwies, denn jeder, aber wirklich jeder verwechselt ständig diese beiden Begriffe.

Das `"public"` besagt, daß die abgeleitete Klasse sich ihrer Herkunft nicht schämt und deshalb die Basisklasse in ihrem öffentlichen Teil liegen soll. Man könnte hier natürlich auch `"private"` oder `"protected"` angeben, oder auch nichts - der Default ist `"private"`.

Hinter der Basisklassen-Deklaration folgt eine ganz normale Strukturdefinition, welche die Eigenschaften beschreibt, die die Klasse `"Entwickler"` zusätzlich zu ihren ererbten haben soll.

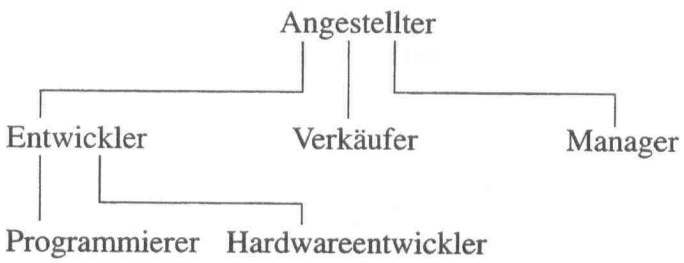
Analog leiten wir die Klassen `"Verkaeufer"` und `"Manager"` ab:

```
class Verkaeufer : public Angestellter
{ public:
     int Umsatz;    // Damit wir wissen, wen wir feuern müssen
     int Ort;         // Nummer der Vertriebs-Aussenstelle
};

class Manager : public Angestellter
{ public:
     char KFZ[10]; // Kennzeichen des Dienstwagens
     int Spesenkonto;
};
```

Das Spielchen läßt sich beliebig weiter treiben: Von `"Entwickler"` leiten wir `"Programmierer"` ab.

```
class Programmierer : public Entwickler
{ public: // Bitfeld mit den Sprachen, die er beherrscht
     struct
       { int C:1, Cplusplus:1, Assembler:1, Smalltalk:1 }
       Sprachen; };
```

Die Klasse für die Lötkolben-Aristen erspare ich mir hier, es dürfte wohl auch so klargeworden sein, wie das mit dem Erben und Ableiten geht. Das Resultat ist eine Klassenhierarchie, die man gern mit so komischen Bäumen (komisch deshalb, weil die Wurzel oben ist) veranschaulicht:



Hübsch, nicht? Jedenfalls schulde ich Ihnen noch ein paar Worte darüber, was das Erben von Eigenschaften in der Praxis heißt.

Überall da, wo eine Basisklasse stehen soll, darf auch eine abgeleitete Klasse benutzt werden, z. B.

```
void main()
{ Programmierer P1;

  P1.Sprachen.Cplusplus = 1;

  P1.Gehalt = 5432;        // Eigentlich Member von "Angestellter"

  Entwickler E1 = P1;   // "P1" wird als "Entwickler" betrachtet
                              // und kopiert
}
```

Wie Sie sehen, ist das mit der Vererbung gar nicht so tiefsinnig, wie man manchmal glaubt. Natürlich werden nicht nur die Daten-Member, sondern auch die Member-Funktionen vererbt:

```
class S
{ int i;
  public:
    int get_i();
};

int S::get_i()
{ return i;
}

class T: public S
{ // usw.
};

void main()
```

```
{ T t1;
  int j = t1.get_i();
}
```

Dabei ist es natürlich nicht erforderlich, `"T::get_i"` noch einmal zu definieren, denn es wird wieder `"S::get_i"` verwendet.

3.2.1.2 Mehrfach-Vererbung und andere Features

Ich erwähnte ja bereits, daß die Symbole `"struct"` und `"class"` weitgehend identisch sind. Folglich kann man durchaus auch mit Strukturen vererben, allerdings ist bei `"struct"` der Zugriffsmodus per Default `"public"`, so daß man das nicht mehr extra angeben muß. Unionen und Bitfelder können dagegen keine Basisklassen besitzen:

```
struct S
{ int i, j;
};

struct T : S
{ int k:4, l:7;
};                  // ERROR

union U : S    // ERROR
{ char c; double d;
};
```

Außerdem muß jede Klasse wirklich definiert sein, bevor man von ihr etwas ableitet:

```
class C;

class D: C        // ERROR
{                     // usw.
};
```

Dadurch ist auch ausgeschlossen, daß zwei Klassen voneinander abgeleitet werden, was ja auch wenig sinnvoll ist.

Zu den neuen Features des 2.0-Standards gehört die Mehrfach-Vererbung. Wie der Name schon andeutet, wird dabei eine Klasse von mehreren Basisklassen gleichzeitig abgeleitet, etwa so:

```
class C1
{ int c1;
};

class C2
{ int c2;
};

class C: public C1, private C2
{ int c;
};
```

Jede Klasse kann beliebig viele Basisklassen haben, aber jede nur einmal:

```
class C { };

class D: C, C    // ERROR
{ };
```

Die Zugriffs-Spezifikationen beziehen sich immer nur auf eine einzelne Basisklasse: Bei

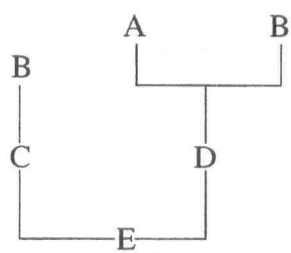
```
class D: public B, C {};
```

hat `"C"` den Default-Modus, also `"private"`.

Mehrfach-Vererbung funktioniert zunächst auch so, wie man es erwartet, allerdings ist einiges komplizierter als bei Einfach-Vererbung. Man kann jetzt wieder einen Baum malen, der diesmal aber keine Klassenhierarchie, sondern die Basisklassen einer einzelnen Klasse darstellt:

```
struct A { int ia; };
struct B { int ib; };
struct C: B { int id;};
struct D: A, B { };
struct E: C, D { int ie;  };
```

ergibt folgenden Graphen für die Klasse `"E"`:



Wie Sie unschwer erkennen, tritt die Klasse `"B"` hier doppelt auf, und das ist dann in der Realität auch so. Der Compiler weiß jetzt nicht mehr, was er tun soll, wenn von der Klasse `"E"` aus auf den Member `"ib"` zugegriffen wird, denn den gibt es ja gleich zweimal:

```
void main()
{ E e1;
  e1.ib = 42;    // ERROR
}
```

Außerdem kann es ja auch vorkommen, daß zwei Basisklassen Member mit identischen Namen vererben:

```
class C1
{ public:
    int x;
};
```

```
class C2
{ public:
     char x;
};

class Der: public C1, public C2
{ double usw;
};

void main()
{ Der d;
  d.x = 0;     // ERROR: welches "x"?
}
```

Dem kann man aber durch Qualifizierungen abhelfen:

```
void main()
{ Der d;
  d.C1::x = 0;
}
```

Wie Sie sehen, bereitet Mehrfach-Vererbung bisweilen seltsame Probleme.

3.2.1.3 Vererbung und Typregeln

Wie bereits angedeutet, kann eine abgeleitete Klasse überall dort stehen, wo ein Element einer ihrer Basisklassen erwartet wird. Das trifft auch auf Pointer-Typen zu: Bei dem Beispiel

```
class B
{ int b;
};

class C : public B
{ int c;
};

B b1;
C c1;
```

kann ein Zeiger auf `"C"` ohne weiteres in einen Zeiger auf `"B"` umgewandelt werden:

```
B* bp = &c1;
```

Vernünftigerweise wird garantiert, daß `"bp"` hier auf den Teil von `"c1"` zeigt, der der Basisklasse entspricht, also wirklich ein `"B"` ist. In der umgekehrten Richtung geht das im Prinzip auch, allerdings muß man hier casten:

```
C* cp = (C*) &b1;
```

In dieser Form ist das natürlich hanebüchen, denn `"b1"` ist ja kein Objekt der Klasse `"C"`, und besitzt insbesondere keinen Member `"c"`. Trotzdem ist diese Typkonvertierung erlaubt, denn oft ist

sie auch sinnvoll, etwa bei der Rück-Umwandlung von Pointern. Wenn das Programm sich sicher ist, daß der Zeiger `"bp"` nicht nur auf ein `"B"`, sondern auf den `"B"`-Teil eines `"C"` zeigt, kann man ihn wieder umwandeln,

```
C* cp2 = (C*) bp;
```

wobei wieder garantiert wird, daß diese Konvertierung sinnvoll und richtig geschieht, d. h. wenn `"bp"` wirklich auf den `"B"`-Teil eines `"C"`-Objekts zeigt, ist `"cp2"` anschließend ein Zeiger auf dieses Objekt der Klasse `"C"`.

Natürlich geht das entsprechend auch über mehrere „Generationen", d. h. ein Zeiger auf eine Klasse kann direkt in eine Basisklasse der Basisklasse ihrer Basisklasse verwandelt werden, und mit einem Cast geht es entsprechend umgekehrt. Natürlich wird auch hier bei Mehrfach-Vererbung geprüft, ob die Basisklasse eventuell doppelt vorkommt:

```
struct A {  int ai;};
struct B : A {int bi;};
struct C : A {int ci;};
struct D : B, C {  int egalwas;  };

D d;
A *ap1 = &d;          // ERROR: "A" kommt in "D" mehrfach vor
A *ap2 = (B*) &d;  // So geht's
```

Die letzte Initialisierung ist möglich, weil hier der Zeiger auf `"d"` zuerst in einen Zeiger auf `"B"` umgewandelt wird, was eine eindeutige Operation darstellt, und erst dann nach `"A*"` konvertiert, was ebenfalls eindeutig ist. Man muß dem Compiler also manchmal sagen, wie er sich durch einen Klassen-Baum hangeln soll.

Völlig analog gilt das oben Gesagte für Referenzen:

```
class B
{ int b; };

class C : public B
{ int c; };

B b1;
C c1;

B& br = c1;
C& cr = (C&) b1;
```

Schon in 3.2.1.1 wurde darauf hingewiesen, daß man Objekte auch direkt in ihre Basisklassen verwandeln kann, z. B.

```
class B
{ int b; };

class C : public B
```

```
{ int c; };

C c1;
B b1 = c1; // Nimm' den B-Teil von c1 und weise ihn b1 zu
```

Es versteht sich, daß bei Mehrfach-Vererbung auch hier geprüft wird, ob die Basisklasse eindeutig ist.

Der umgekehrte Fall,

```
C c2 = (C) b1;   // ERROR
```

ist glücklicherweise nicht erlaubt, denn hier weiß der Compiler, daß `"b1"` ein `"B"`-Objekt ist und garantiert nicht zu einem `"C"`-Objekt gehört.

3.2.2 Scoperegeln und Zugriffskontrolle bei Vererbung

3.2.2.1 Noch mehr Details

Bekanntlich hat jede Klasse ihren eigenen Scope, und das gilt auch bei Vererbung. Eine abgeleitete Klasse kann deshalb problemlos Namen aus ihren Basisklassen überdecken:

```
class Base
{ public: int member;
};

class Derived : Base
{ public: void member(char*);
};
```

Im Zweifelsfall kann man wie gewohnt durch Qualifizierungen klarstellen, was gemeint ist:

```
void main()
{ Derived d;
  char *str = "Niemand erwartet die spanische Inquisition!";

  d.Base::member = 26731;
  d.Derived::member(str); // "d.member" hätte hier auch gereicht
}
```

Trotzdem gibt der Compiler beim obigen Programm eine Fehlermeldung aus, denn er ist der Meinung, daß die Funktion `"main"` nicht auf den Namen `"member"` aus `"Base"` zugreifen darf. Das kommt etwas unerwartet, denn schließlich ist `"member"` in `"Base"` doch eigens als `"public"` deklariert worden. Des Rätsels Lösung ist, daß `"Base"` eine private Basisklasse von `"Derived"` ist (Sie erinnern sich: hinter `"class"` ist der Zugriffsmodus defaultmäßig `"private"`). Das bedeutet, daß alle Member aus der Basisklasse `"Base"` in `"Derived"` privat sind, ja sogar die auf Basisklasse als solche von außen nicht zugegriffen werden kann:

```
Derived d;
Base * bp = &d;     // ERROR: Basisklasse ist privat!
```

Dieses Konzept ist auf den ersten Blick zwar eigenartig, aber doch schlüssig: Wenn der Umstand, daß eine Klasse von einer anderen abgeleitet ist, zum implementatorischen Teil gehört, ist es natürlich sinnvoll, die Basisklasse zu „verstecken" und jeden Zugriff von der Benutzerseite zu unterbinden.

3.2.2.2 Der Zugriffsmodus „protected"

Bisher (und das war auch im „1.0-Standard" so) haben wir nur zwischen den internen Bereichen einer Klasse (`"private"`) und der Schnittstelle zur Außenwelt (`"public"`) unterschieden. Abgeleitete Klassen gehören in diesem Sinne zur „Außenwelt" ihrer Basisklassen, denn eine abgeleitete Klasse kann natürlich nicht auf die privaten Member ihrer Basisklasse zugreifen. Das ist sinnvoll, aber nicht wirklich befriedigend. Kurz gesagt, es entstand im Laufe der Zeit das Bedürfnis, einen Unterschied zwischen abgeleiteten Klassen und Code, der wirklich außerhalb der Klasse liegt, zu differenzieren. Das Ergebnis dieser Überlegungen war der Zugriffsmodus `"protected"`, der dann in den „2.0-Standard" aufgenommen wurde.

`"protected"` wird genau wie `"public"` und `"private"` benutzt. Auf einen Bezeichner mit diesem Zugriffsmodus können abgeleitete Klassen zugreifen, sonst gilt er als `"private"`. Ein Beispiel:

```
class Klasse
{ protected:
    int imember;
};

class Abgeleitet : public Klasse
{ void set(int i)
  { imember = i;            // OK, da abgeleitete Klasse
  }
};

void main()
{ Klasse k;
  k.imember = 3718847;     // ERROR
}
```

Das ist ein ganz nützliches Feature, wenn man Klassen hat, die mehr oder weniger ausschließlich dafür vorgesehen sind, daß von ihnen weitere Klassen abgeleitet werden.

Da wir jetzt schon drei Zugriffsarten haben, ist es wohl an der Zeit, noch einmal klarzustellen, welche Auswirkungen der Zugriffsmodus einer Basisklasse hat:

Wenn die Basisklasse, wie oben geschehen, als `"public"` deklariert wird, darf von jeder Programmstelle auf die Basisklasse zugegriffen werden, d. h. die Eigenschaft, daß die Klasse von einer anderen abgeleitet ist, ist kein implementatorisches Detail, sondern öffentlich. Natürlich behalten die Member, die als `"private"` oder `"protected"` deklariert sind, diesen Modus auch bei.

Eine Basisklasse, die `"private"` ist, ist auch nicht weiter schwer zu verstehen: Alle Member bzw. Basisklassen dieser Basisklasse sind absolut privat, sogar die Tatsache, daß abgeleitet wurde, ist für jeden Zugriff von außen tabu, so daß man z. B. keinen Zeiger auf die Klasse in einen Zeiger auf die

Basisklasse verwandeln kann - nur die abgeleitete Klasse selbst (bzw. ihre Member-Funktionen und Friends) dürfen das. Und wenn last not least der Zugriffsmodus der Basis `"protected"` ist, hängt es von Standpunkt des Betrachters ab, ob `"private"` oder `"public"` gilt: In der abgeleiteten Klasse wirkt diese Basisklasse wie `"public"`, für Klassen, die wiederum davon abgeleitet werden, bleibt der Zugriffsmodus `"protected"`, und für alles, was nicht zu diesen Klassen oder ihren Friends gehört, ist keinerlei Zugriff auf die Basisklasse erlaubt, so als wäre sie `"private"`. Aber im Grunde genommen hätte ich mir diese ganzen Regeln auch sparen können, denn wenn man einmal scharf darüber nachdenkt, ist die ganze Zugriffsregelung intuitiv völlig klar.

Daran ändert auch das folgende Faeture nichts wesentliches:

3.2.2.3 Zugriffs-Deklarationen

Für manche Sicherheitsfetischisten (Vertrauen ist gut, Kontrolle besser) sind die bisherigen Regelungen für die Zugriffsrechte noch nicht umfangreich genug, und deshalb gibt es auch für sie ein nettes kleines Spielzeug: Zugriffs-Deklarationen für abgeleitete Klassen.

Man stelle sich folgende Situation vor: Sie wollen von einer Klasse ableiten und die Basisklasse als `"private"` oder `"protected"` vor unbefugten Zugriffen schützen, aber trotzdem den einen oder anderen Member aus der Basisklasse öffentlich zugänglich machen. Das folgende Programm illustriert, wie das geht:

```
class Bas
{ private:
    int a;
  protected:
    int b;
  public:
    int c, d;
};

class Der : private Bas
{ public:
    int e;
    Bas::c;
};

void main()
{ Der x;
  x.c = 42;      // Geht, da in "Der" public
  x.d = 4711;    // ERROR, private Basisklasse
}
```

Der Zugriff auf `"d"` über `"Der"` ist erwartungsgemäß verboten, denn die Basisklasse `"Bas"` ist privat. Dagegen kann `"c"` auch über `"Der"` benutzt werden, denn er wird dort ausdrücklich als öffentlich deklariert.

Wie Sie sehen, erfolgt eine solche Zugriffs-Deklaration ganz einfach über den Namen des Members ohne jegliche Typangabe. Der neue Zugriffsmodus muß derselbe sein, den der Member auch in der Basisklasse hatte, denn alles andere ist ziemlich sinnlos:

```
class B1
{ private:
    int i;
  public:
    int j;
};

class C1 : private B1
{ public:
    B1::i;      // ERROR
};

class C2: public B1
{ private:
    B1::j;      // ERROR
};
```

Der erste Fehler ist klar, man kann den Schutz eines Members nicht so ohne weiteres verringern, denn das würde ja das Ganze System der Zugriffs-Kontrolle ab absurdum führen, nach dem Motto: Wenn ein Member privat ist, leitet wir einfach eine Klasse ab und setzen uns die Rechte so, wie wir sie haben wollen. Der zweite Fehler ist auf den ersten Blick nicht so offensichtlich, denn was soll schlecht daran sein, einen Member sicherheitshalber privat zu machen? Der Haken ist nur, daß das ganz einfach sinnlos ist: Wenn man einen Objekt der Klasse `"C2"` hat, nimmt man einfach einen Zeiger darauf, wandelt diesen in einen Zeiger auf `"B1"` ab (geht, denn schließlich ist `"B1"` hier öffentliche Basisklasse) und greifen darüber auf `"j"` zu - alles null Problemo.

Da der Zugriffsmodus also nicht wirklich verändert werden kann, folgt daraus, daß Zugriffs-Deklarationen bei öffentlichen Basisklassen (wie `"C2"` im obigen Beispiel) nutzlos sind, denn hier behalten alle Member der Basisklasse ohnehin ihren alten Modus.

Einen eigentümlichen Effekt kann man bei Zugriffs-Deklarationen für überladene Funktionen beobachten: Da die Rechte hier ausschließlich über den Namen gesetzt werden, müssen alle Funktionen dieses Namens denselben Zugriffsmodus haben:

```
class Basis
{ private:
    int g();
    int g(char*);
    void f(int);
  public:
    void f(char);
};

class Klasse : private Basis
{ private:
    Basis::g;      // OK, beide "private"
```

```
  public:
    Basis::f;      // ERROR
};
```

Durch die "public"-Deklaration würden beide "f"-Funktionen in "Klasse" als öffentlich deklariert werden, aber das gibt natürlich Ärger, denn "f(int)" war ja vorher privat. Auch eine Deklaration als "private" würde nicht helfen, denn dann würde wiederum der Zugriffsmodus von "f(char)" verändert. Zugriffs-Deklarationen für überladene Funktionsnamen sind also nur erlaubt, wenn alle Funktionen (wie "g") den gleichen Zugriffsmodus haben.

3.2.3 Virtuelle Member-Funktionen

3.2.3.1 Ein Beispiel

Lassen wir uns zusammenfassen, was wir bisher an objektorientierten Features kennengelernt haben: Wir können Funktionen an Strukturen anbinden, haben eine Möglichkeit festzulegen, welcher Programmteil auf welchen Member zugreifen darf, und können Strukturtypen durch Vererbung erweitern oder auch zusammenfassen. Das ist alles sehr nützlich, aber irgendwie doch noch nicht so besonders schlau. Aber das kommt jetzt, aber hallo.

Angenommen, Sie haben einen Zeiger auf ein Objekt einer bestimmten Klasse. Dann wissen Sie ja bereits, daß dieser Zeiger in Wirklichkeit durchaus auf ein Objekt einer ganz anderen Klasse zeigen kann. Vielleicht erinnern Sie sich noch an das allererste Beispiel aus Abschnitt 3.2.1.1, an die Abgestellten und Manager und so weiter. Nehmen wir an, die einzelnen „Angestellten" sind in einer Liste angeordnet. Nun haben wir einen Zeiger auf ein solches Objekt und wollen gerne die Daten ausgeben, und zwar vollständig, und dazu müssen wir wissen, von welcher Klasse ein Objekt wirklich ist, also ob wir es mit einem Programmierer, Manager oder bloß einem einfachen Angestellten zu tun haben.

Es wird Sie vielleicht enttäuschen, daß es in C++ keine direkte Möglichkeit gibt, genau dieses festzustellen, denn dann müßte bei wirklich jedem Objekt in irgendeiner Form dessen Typ abgespeichert werden, und das ist erstens gar nicht so einfach und würde zweitens Speicherplatz kosten. Aber das ist eigentlich nicht so schlimm, denn normalerweise muß man nicht wirklich wissen, mit welcher Klasse man es zu tun hat, sondern nur, welche momentan relevanten Eigenschaften diese hat.

So, da Sie jetzt offenbar den vorherigen Satz in seiner ganzen Bedeutungsschwere verarbeitet haben, ist es langsam wieder an der Zeit, konkret zu werden (wer „A" sagt, muß auch „ngst" sagen). Also der Schlüssel dazu sind virtuelle Memberfunktion.

Eine virtuelle Memberfunkion sieht zuerst wie eine ganz normale Funktion aus, nur daß sie mit dem Schlüsselwort "virtual" deklariert wird:

```
#include <stream.h>

class Angestellter
{ public:
    char Vorname[20], Nachname[20];
```

```
    unsigned int Gehalt;
    virtual void Ausgabe();
};

void Angestellter::Ausgabe()
{ cout « Vorname « " " « Nachname « "\n";
  cout « "Gehalt: " « Gehalt « " DM\n";
}
```

Ganz normal ist diese Funktion aber keineswegs: In Wirklichkeit trägt der Compiler jetzt bei jedem Objekt der Klasse `"Angestellter"` einen Vermerk ein, daß immer dann, wenn auf diesem Objekt die Funktion `"Ausgabe"` aufgerufen wird, die Funktion `"Angestellter::Ausgabe"` genommen werden soll (vorerst können Sie sich das so vorstellen, als würde in jedes Objekt automatisch ein Zeiger auf diese Funktion eingetragen - die Realität ist ein wenig komplizierter, aber im Prinzip ist das so). Jetzt werden Sie sich fragen, was denn wohl sonst aufgerufen werden soll, wenn nicht `"Angestellter::Ausgabe"`? Nun, wir leiten jetzt die Klasse „Entwickler" ab, die eine eigene Ausgabefunktion desselben Namens tragen soll:

```
class Entwickler : public Angestellter
{ public:
    char Projekt[25];
    void Ausgabe();
};

void Entwickler::Ausgabe()
{ Angestellter::Ausgabe();
  cout « "arbeitet an: " « Projekt « "\n";
}
```

Die Klasse `"Entwickler"` hat ihre eigene Funktion `"Ausgabe"`, die zunächst die `"Ausgabe"`-Funktion aus der Basisklasse aufruft und dann noch den Namen des Projekts ausgibt. So weit scheint alles noch völlig normal, aber durch den Umstand, daß die Funktion `"Ausgabe"` denselben Namen trägt wie eine virtuelle Funktion in der Basisklasse, ist sie automatisch ebenfalls virtuell (man hätte hier genau so gut auch ein `"virtual"` hinzufügen können). Also erhält auch jedes Objekt der Klasse `"Entwickler"` einen Zeiger (bzw. so etwas ähnliches) auf die Funktion `"Entwickler::Ausgabe"`. Aber jetzt kommt der Trick: In Objekten der Klasse `"Entwickler"` wird auch der entsprechende Zeiger innerhalb ihrer `"Angestellter"`-Basisklasse auf diese Funktion gesetzt! Nehmen wir also an, wir haben einen Zeiger auf `"Angestellter"`, z. B. als Parameter einer Funktion:

```
void f(Angestellter *a)
{  a->Ausgabe();
}
```

Beim Aufruf `"a->Ausgabe"` hat der Compiler natürlich keine Ahnung, ob `"a"` nun auf einen Angestellten oder einen Entwickler zeigt. Deshalb schaut er nach, auf welche Funktion dort der Zeiger für die Funktion `"Ausgabe"` zeigt, und springt dahin, also entweder nach `"Angestellter::Ausgabe"` oder nach `"Entwickler::Ausgabe"`. Ohne virtuelle Funktionen wären die

„wahre" Klasse des Objekts `"*a"` ignoriert und ganz einfach `"Angestellter::Ausgabe"` ausgeführt worden.

3.2.3.2 Die Details

Es wurde oben behauptet, in C++ könne man nicht direkt feststellen, von welcher Klasse ein Objekt wirklich ist, wenn man bloß einen Zeiger bzw. eine Referenz darauf besitzt. Scheinbar hat der Compiler bei virtuellen Funktionen aber doch diese Möglichkeit. Was ist denn nun?

Bei einer virtuellen Funktion weiß das Programm keineswegs, mit welcher Klasse es zu tun hat, sondern besitzt lediglich Informationen darüber, welche Funktion aufzurufen ist, also, wie ich es oben bereits so schön schwülstig formulierte, welche dazu relevanten Eigenschaften das Objekt hat. Was der Unterschied ist? Nun, eine virtuelle Funktion muß keineswegs auf jeder nachfolgenden Stufe der Klassenhierarchie überdeckt werden:

```
class A
{ public:
     int ai;
     virtual void f();
};

class B: public A
{ int bi;};
```

Hier wird natürlich immer `"A::f"` aufgerufen, denn `"B::f"` gibt es ja nicht. Wenn wir weiter ableiten,

```
class C: public B
{ public:
     char ci;
     void f();
};

class D: public C
{   double dd; };

A *ap;
```

gibt es vier Klassen, aber nur zwei Möglichkeiten, was bei `"ap->f()"` aufgerufen wird: entweder `"A::f"` (bei Objekten der Klassen `"A"` und `"B"`) oder `"C::f"` (Klassen `"C"` und `"D"`). Also wird die Wahl der richtigen virtuellen Funktion beim Aufruf nicht anhand der wirklichen Klasse, sondern anhand der höchsten Klasse der Hierarchie, in der die Funktion überdeckt wird, getroffen. Das mag spitzfindig klingen, ist aber der Hauptunterschied in den objektorientierten Konzepten von C++ und Objective C. In Objective C kann man feststellen, von welcher Klasse ein Objekt ist, was in C++ nur mit hohem Aufwand möglich ist, dafür ist C++ wesentlich schneller und braucht weniger Speicherplatz.

Wenn Sie sich die Definition der Funktion `"Entwickler::Ausgabe"` noch einmal ansehen, werden Sie sich vielleicht fragen, warum es hier nicht zu einer endlosen Rekursion kommt, schließlich

wird doch die virtuelle Funktion **"Angestellter::Ausgabe"** aufgerufen, und die ist bei Entwicklern durch deren Ausgabefunktion überdeckt. Dahinter steckt eine recht sinnvolle Definition: Beim Funktionsaufruf über einen qualifizierten Bezeichner wird diese Funktion grundsätzlich als nicht-virtuell betrachtet, hier also wirklich **"Angestellter::Ausgabe"** aufgerufen, wie es beabsichtigt war.

Virtuelle Funktionen müssen durchaus nicht immer den gleichen Zugriffsmodus haben:

```
class Base
{ public:
    virtual int f(int);
};

class Derived : public Base
{ private:
    int f(int);
};
```

Sonderlich sinnvoll ist das zwar nicht, aber es ist halt so definiert.

Eine Memberfunktion überdeckt eine virtuelle Funktion einer Basisklasse nur dann, wenn Parameterliste und Ergebnistyp exakt übereinstimmen:

```
class B
{ public:
    virtual int f(char*);
};

class C : public B
{ public:
    unsigned f(unsigned char*);
};
```

Hier ist **"C::f"** nicht virtuell, denn sie unterscheidet sich in Parameterliste und Ergebnistyp „etwas" von **"B::f"**. Wenn man der Deklaration von **"C::f"** ein **"virtual"** vorangestellt hätte, wäre sie zwar virtuell, würde **"B::f"** aber nicht überdecken.

Eine virtuelle Funktion kann nicht statisch sein, denn beim Aufruf muß anhand eines konkreten Datenobjekts festgestellt werden, welche Funktion zu benutzen ist. Wird sie als **"inline"** deklariert, muß trotzdem Code für sie erzeugt werden (denn ein Zeiger auf die Funktion muß in die jeweiligen Objekte eingetragen werden) und in der Regel erfolgt der Funktionsaufruf auch nicht über diesen Zeiger und nicht über **"inline"**. Aber MaxonC++ optimiert virtuelle Funktionsaufrufe, bei denen der Objekt-Typ und damit die gewünschte Funktion schon zur Compile-Zeit feststeht, zu gewöhnlichen Funktionsaufrufen:

```
class C
{ int i;
  public:
    virtual void f()
    { i = 42; }
```

```
};

void main()
{ C c1;
  c1.f();              // Nur "C::f" möglich, also kein
                           // Virtueller Aufruf nötig
}
```

Virtuelle Datenmember gibt es übrigens nicht:

```
class Obskur
{ virtual int i;   // ERROR };
```

Falls man so etwas wirklich brauchen sollte, kann man sich mit virtuellen Memberfunktionen, die jeweils eine Referenz auf den passenden Member zurückgeben, behelfen.

3.2.3.3 Abstrakte Klassen

Es gibt Situationen, bei denen man eine Klasse ausschließlich als Basisklasse für andere Klassen benutzen will und nicht die Absicht hat, Objekte dieser Klasse zu erzeugen. Dies kommt insbesondere bei abstrakten Datentypen vor, z. B. kann man sich eine Klasse **"Menge"** erzeugen und erst durch weiteres Ableiten eine wirkliche Menge mit „echten" Elementen erzeugen. Wenn man aber eine Klasse nie wirklich zum Erzeugen von Objekten, sondern nur zum Vererben benutzen will, wäre es natürlich überflüssig, die virtuellen Funktionen der Basisklasse wirklich zu definieren.

Deshalb hat man im 2.0-Standard „abstrakte Klassen" hinzugefügt. Das Schlüsselkonzept dazu sind die „völlig virtuellen Funktionen".

Nehmen wir an, wir haben unsere Klasse **"Mengenelement"**, die irgendein Element in irgendeiner Menge realisieren soll. Auch hier braucht man, wie so oft, eine Funktion zur Ausgabe dieses Mengenelements, und da wir nichts über den konkreten Mengentypen, den wir später ableiten wollen, wissen, deklarieren wir diese Memberfunktion sinnvollerweise als virtuell.

```
class Mengenelement
{ // Irgendwelche internen Daten, z. B. Listenstruktur
  public:
    virtual void Ausgabe() = 0;
};
```

Das **"= 0"** hinter der Funktionsdeklaration besagt, daß es sich hier um eine „völlig virtuelle" Funktion handelt, die wir nie zu definieren gedenken, weil es ganz einfach sinnlos ist, ein Mengenelement auszugeben, bei dem wir noch nicht einmal wissen, von welchem Typ die Daten der Menge sind. Dadurch, daß die Klasse eine völlig virtuelle Funktion besitzt, ist sie für den Compiler eine abstrakte Klasse, er wird nicht zulassen, daß wir ein Objekt dieser abstrakten Klasse erzeugen:

```
void falsch()
{ Mengenelement e1;              // ERROR
  Mengenelement *ep;            // OK
  ep = new Mengenelement; // ERROR
}
```

Aus unserem abstrakten Datentypen erzeugen wir jetzt eine ganz bestimmte Menge, nämlich eine Zahlenmenge, und zu diesem Zwecke brauchen wir den entsprechenden Elementtyp:

```
class ZahlenElement : public Mengenelement
{ public:
     int zahl;           // Das eigentliche Mengenelement
     void Ausgabe();
};

void ZahlenElement::Ausgabe()
{ cout « zahl;
}
```

In der Klasse **"ZahlenElement"** wird die virtuelle Funktion **"Ausgabe"** aus der Basisklasse durch eine (implizit zwar virtuelle, aber nicht völlig virtuelle) Funktion überdeckt. Dadurch besitzt die Klasse **"ZahlenElement"** keine völlig virtuellen Funktionen mehr und ist auch nicht mehr abstrakt. Also können wir Objekte der Klasse **"ZahlenElement"** nach Herzenslust und ungeniert erzeugen.

Wird von einer abstrakten Klasse eine neue Klasse abgeleitet, ohne alle völlig virtuellen Funktionen der Basisklasse zu überdecken, gilt die neue Klasse ebenfalls als abstrakt. Man kann natürlich davon weiter ableiten und irgendwann auch die letzte völlig virtuelle Funktion richtig deklarieren, um eine „verwertbare" Klasse zu erhalten.

3.2.4 Ein paar Anmerkungen zur Implementation

Womöglich sind Sie jetzt völlig verwirrt und fragen sich, was für mystische Dinge ein C++-Programm wohl tut, um alle diese abgefahrenen objektorientierten Features zu implementieren. Sollte das so sein, kann ich Sie beruhigen - all das ist letzten Endes ziemlich simpel. Schließlich ist der objekt-orientierte Teil von C++ nach dem zentralen Prinzip konzipiert, so daß nirgendwo im Vergleich zu „normalen" C-Lösungen ein Overhead an Laufzeit oder Speicherbedarf entstehen soll, eher im Gegentum. Und genau davon soll jetzt die Rede sein.

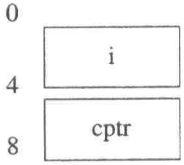
Fangen wir ganz harmlos an: Eine gewöhnliche Struktur besteht, wie Sie sich denken können, aus einem zusammenhängenden Speicherbereich, in dem die Member liegen, und zwar in der Reihenfolge, in der sie deklariert wurden. Daran ändert sich natürlich nichts, wenn man „class" statt „struct" benutzt. Auch unterschiedliche Zugriffsrechte beeinflussen die Abbildung auf das Zielprogramm keineswegs, denn diese Rechte werden bekanntlich bereits vom Compiler geprüft und folglich besteht kein Bedarf, irgendwelche Angaben darüber in die real existierende Struktur einzutragen. Ach ja, statische Member liegen nicht innerhalb eines Datenobjekts, sondern genau einmal sonstwo im Speicher. Die Deklartion dieser Member innerhalb der Klassendefinition sollte Sie nicht darüber hinwegtäuschen, daß es sich hier um ganz normale statische Daten handelt, nur ihre Sichtbarkeit ist auf eine Klasse beschränkt.

Auch bei Vererbung sieht das keineswegs komplizierter aus. Betrachten wir zunächst Einfach-Vererbung, also Ableitung von nur einer Basisklasse. Ich sagte irgendwo oben, daß man sich Vererbung

wie eine Erweiterung einer Struktur vorstellen kann, und genau so realisiert das MaxonC++ auch, und zwar auf die kanonische Weise: Eine abgeleitete Klasse ist immer noch ein zusammenhängender Speicherbereich, wobei der Anfang dieser neuen Struktur der Basisklasse entspricht. Sieht z. B. eine Basisklasse folgendermaßen aus,

```
struct Bas
{ int i;
  char *cptr;
};
```

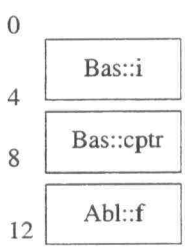
so belegt jedes Objekt dieses Typs acht Bytes, die wie folgt aussehen:



Die Zahlen am linken Rand stellen den Offset innerhalb der Struktur dar, d. h. `"i"` beginnt an der Startadresse der Struktur und `"cptr"` vier Bytes weiter. Wenn wir jetzt eine Klasse ableiten, z. B.

```
struct Abl : public Bas
{ float f; };
```

sieht ein Objekt dieser Klasse folgendermaßen aus:



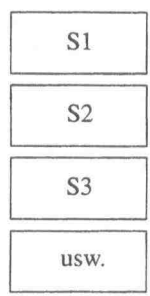
Die Konsequenz ist klar: Wenn man einen Zeiger auf ein solches Objekt hat und ihn als Zeiger auf `"Bas"` benutzt, muß der Compiler nichts weiter tun, denn die Anfangadresse eines `"Abl"`-Objekts ist gleichzeitig die Anfangsadresse eines gültigen Objekts der Klasse `"Bas"`, mit allem was dazu gehört.

Bei Mehrfach-Vererbung ist das etwas komplizierter. Auch hier liegen die Basisklassen am Anfang der abgeleiteten Klasse, aber wie im richtigen Leben kann nur eine die Erste sein. Unter MaxonC++ ist das grundsätzlich die, die zuerst deklariert wurde:

```
struct S1
{ // irgendwas...
};
```

```
struct S2
{ // nochwas...
};

struct S3
{ // egal, was...
};

struct T : public S1, public S2, public S3
{ // usw...
}
```

Ein Objekt der Klasse `"T"` sieht dann so aus (beachten Sie bitte, daß ein Kästchen im folgenden Bildchen nicht wie oben einen einzelnen Member, sondern alle Member einer Klasse darstellt):



Nun hat der Compiler es nicht mehr so leicht: Um einen Zeiger auf `"T"` in einen Zeiger auf `"S2"` oder `"S3"` zu verwandeln, muß er dafür sorgen, daß ein entsprechender Offset addiert wird. Auch bei der umgekehrten Konvertierung (die bekanntlich einen Cast verlangt) muß der entsprechende Offset berücksichtigt, d. h. subtrahiert werden.

Bliebe noch zu erwähnen, wie das mit den virtuellen Funktionen funktioniert. Eine normale, also nicht-virtuelle Member-Funktion ist eine Funktion wie jede andere auch, wobei der `"this"`-Zeiger als versteckter Parameter übergeben wird. Ein Funktionsaufruf der Form

```
xyz.f(p1, p2, p3);
```

veranlaßt den Compiler lediglich, die Adresse des Objekts zu nehmen und als erstes Argument zu verwenden. Es wird also so etwas erzeugt wie

```
IrgendEineKlasse::f(&xyz, p1, p2, p2);
```

Wenn ein Programm aber erst zur Laufzeit feststellen soll, welche konkrete Member-Funktion aufzurufen ist, braucht sie zweierlei: Einen Zeiger auf die Funktion (Zeiger auf Funktionen gibt es auch außerhalb der OO-Welt) und den richtigen Offset für den `"this"`-Pointer. Sehen Sie sich noch einmal die obige Abbildung an: Wenn ein Programm einen Zeiger auf ein `"S2"` hat, das aber in Wirklichkeit nur Bestandteil eines `"T"` ist, und auf diesem `"S2"`-Objekt eine virtuelle Member-Funktion, die in `"T"` überdeckt wird, aufrufen soll, ist der `"this"`-Zeiger zu modifizieren, denn eine Funkti-

on aus `"T"` verlangt nun einmal einen Zeiger auf ein `"T"`, was hier etwas anderes als ein Zeiger auf ein `"S2"` ist. Also muß das Programm nicht nur die Adresse der aufzurufenden Funktion ermitteln, sondern auch noch den richtigen Offset vom `"this"`-Zeiger subtrahieren.

Jede Klasse kann 739 oder noch mehr verschiedene virtuelle Funktionen haben, und damit wäre es wenig sinnvoll, in jedem Objekt für jede virtuelle Funktion einen Zeiger und einen Offset anzubringen. Statt dessen legt MaxonC^{++} für jede benutzte Klassen-Konfiguration eine Tabelle an, in der alle diese Zeiger und Offsets stehen. Ein Zeiger auf diese Tabelle wird automatisch als „unsichtbarer Member" in jedes Objekt eingetragen, sobald es erzeugt wird, z. B. durch eine Variablendeklaration oder mit `"new"`. Später werden wir sehen, daß der Compiler hier einen „Default-Konstruktor" selbsttätig erzeugt und aufruft.

Das ist übrigens auch ein Grund dafür, daß die alte `"malloc"`-Funktion in C^{++} durch `"new"` ersetzt wurde. Der Compiler muß überall dort, wo ein Datenobjekt mit virtuellen Funktionen erzeugt wird, ein paar Zeilen Code erzeugen, die den Zeiger auf die Virtual-Tabelle in das Objekt eintragen. Das kann er aber nur, wenn er weiß, wo welches Objekt erzeugt wird, was mit `"malloc"` nicht möglich ist, denn das ist nur eine Bibliotheksfunktion wie jede andere auch und der Compiler kann nicht wissen, was sie tut und ob sie ein neues Objekt alloziert oder weiß der Geier was. `"new"` ist hingegen fester Bestandteil der C^{++}-Syntax und somit ist für den Compiler erkennbar, welches Datenobjekt davon erzeugt wird.

3.2.5 Virtuelle Basisklassen

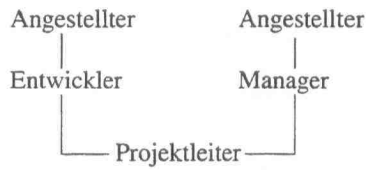
Oben wurde schon angedeutet, welche - bisweilen immense - Probleme das Konzept der Mehrfachvererbung mit sich bringen kann. Hinzu kommt noch, daß man sie auch nicht so oft braucht oder zumindest mit etwas Mühe durch Einfach-Vererbung ersetzen kann. Ein spezielles Problem wurde aber bisher nur angedeutet: Eine Basisklasse kann innerhalb einer abgeleiteten Klasse in mehreren Instanzen auftreten. Dazu zunächst ein kleines Beispiel, wieder einmal so richtig aus dem Leben gegriffen:

Wir betrachten erneut die Sache mit der Belegschaft unserer ominösen Firma, die zunächst einmal aus lauter Angestellten besteht:

```
class Angestellter
{ private:
     // Irgendwelche implementatorische Details
  public:
     char Vorname[20], Nachname[20];
     unsigned int Gehalt;
     // usw..
};
```

Ferner gab es ja auch noch die davon abgeleiteten Klassen `"Entwickler"` und `"Manager"` mit jeweils besonderen Eigenschaften. Zu jedem Projekt gehört aber normalerweise ein Projektleiter, und der soll in unserem Datenmodell die Eigenschaften eines Entwicklers und eines Managers in

sich vereinigen. Bei gewöhnlicher Vererbung sieht die Klasse `"Projektleiter"` dann folgendermaßen aus:



Ei der daus, was ist denn das? Die Basisklase `"Angestellter"`, und mit ihr alle ihre Member, tritt n der Klasse doppelt auf! Das ist natürlich in höchstem Maße unpraktisch, denn erstens werden etzt die Angestellten-Daten eines Projektleiters doppelt abgespeichert und zweitens weiß der Compiler ohne besondere Qualifizierungen oder Typwandlungen nie so recht, welche der beiden Basisklassen er nehmen soll, wenn die Basisklasse `"Angestellter"` eines Projektleiters benutzt wird. Das ist schon ziemlich unangenehm.

Solche Probleme löst man durch virtuelle Basisklassen. Wenn eine Klasse eine virtuelle Basis hat, heiß das so viel wie „irgendwo im Datenobjekt will ich so eine Basisklasse haben, aber möglichst nur einmal, und es stört mich nicht, wenn ich mir die Basisklasse mit anderen Klassen teilen muß". Also leiten wir die Klassen `"Manager"` und `"Entwickler"` virtuell von `"Angestellter"` ab:

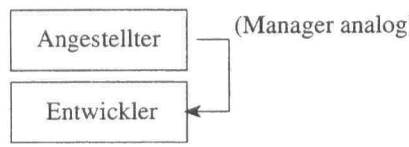
```
class Entwickler : virtual public Angestellter
{ public:
     char Projekt[25];
};

class Manager : virtual public Angestellter
{ public:
     char KFZ[10];    // Kennzeichen des Dienstwagens
     int Spesenkonto;
};
```

Es ist übrigens Banane, ob Sie `"virtual public"` oder `"public virtual"` schreiben. In beiden Fällen bezieht sich das `"virtual"` (genau wie der Zugriffsmodus) ausschließlich auf die nächste folgende Basisklasse, so daß man bei Mehrfach-Vererbung ggf. vor jede Basisklasse ein `"virtual"` schreiben muß.

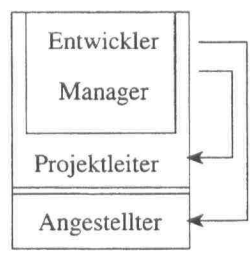
Wenn wir nun den `"Projektleiter"` von `"Entwickler"` und `"Manager"` ableiten, können wir durchaus beide Basisklassen erneut als virtuell deklarieren. Um die ganze Angelegenheit nicht unnötig zu komplizieren, ersparen wir uns das zunächst aber:

```
class Projektleiter: public Entwickler, public Manager
{ public:
     int Bonus;     // Prämie, die er bekommt, falls sein Projekt
               // ausnahmsweise mal termingerecht fertig wird
};
```

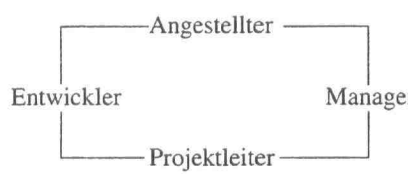
Eine Klasse mit virtueller Basisklasse hat an genau der Stelle, wo eigentlich diese Basisklasse stehen würde, lediglich einen Zeiger auf eine solche Basis. Beim Zugriff auf die virtuelle Basisklasse muß das Programm sich nur an diesem Pointer entlanghangeln. Die virtuelle Basisklasse könnte prinzipiell irgendwo im Speicher stehen, aber sinnvollerweise legt MaxonC++ sie direkt hinter der abgeleiteten Klasse ab, so daß das Objekt einen zusammenhängenden Speicherbereich belegt. Ein Objekt der Klasse „Entwickler" sieht also folgendermaßen aus:



Bei der Klasse `"Projektleiter"` können wir das Spielchen fortgesetzt: Die beiden Basisklassen `"Entwickler"` und `"Manager"` besitzen jeweils einen Zeiger auf ein `"Angestellter"`-Objekt, das irgendwo am Ende der Klasse steht, aber beide verweisen auf dasselbe Objekt:



Als Ableitungsbaum sieht das so aus:



Anhand dieser über alle Maßen anschaulichen Grafiken wird wohl klar, wie es kommt, daß es in `"Projektleiter"` nur einen einziges `"Angestellter"` gibt und jeglicher Zugriff darauf eindeutig ist, auch wenn der Compiler auf dem Weg von `"Projektleiter"` nach `"Angestellter"` die freie Auswahl hat, ob er den Zeiger aus `"Manager"` oder den aus `"Entwickler"` nehmen will. Das Eintragen dieser Zeiger geschieht übrigens genauso wie das der Zeiger auf die Virtual-Function-Tables, nämlich vollautomatisch, sobald ein neues Objekt erzeugt wird.

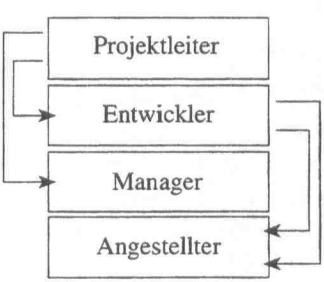
Somit können wir getrost auf alle Member der Klasse `"Projektleiter"` zugreifen, den Rest macht schon der Compiler:

```
#include <string.h>
#include <stream.h>

void main()
{ Projektleiter P1;

   strcpy(P1.Vorname, "Großer");
   strcpy(P1.Nachname, "Guru");
   P1.Gehalt = 26731;
   strcpy(P1.Projekt, "Grober Unfug");
   strcpy(P1.KFZ, "PB-H 37");
   P1.Spesenkonto = 4711;
   P1.Bonus = 666;

   // mach' irgendwas

   cout « P1.Vorname « " " « P1.Nachname « "\n";
   cout « "Einkommen:    " « P1.Gehalt « " DM\n";
   cout « "Projekt:      " « P1.Projekt « "\n";
   cout « "Dienstwagen: " « P1.KFZ « "\n";
   cout « "Spesen:       " « P1.Spesenkonto « " DM\n";
   cout « "Prämie:      " « P1.Bonus « " DM\n";
}
```

Just for fun überlegen wir uns jetzt noch, wie die Klasse aussehen würde, wenn `"Entwickler"` und `"Manager"` ebenfalls virtuelle Basiskklassen von `"Projektleiter"` wären:

```
class Projektleiter: virtual public Entwickler,
       virtual public Manager ...
```

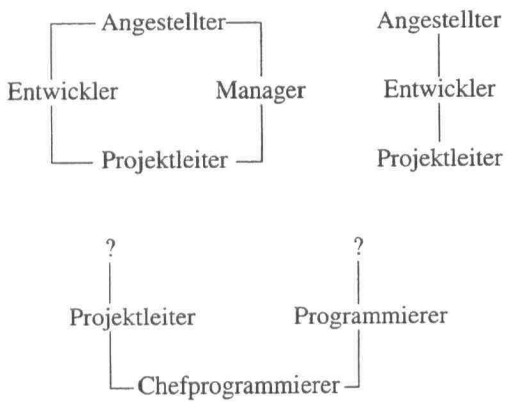
Nun hat die Klasse `"Projektleiter"` da, wo vorher ihre beiden Basisklassen hatte, nur zwei Zeiger auf diese Objekte, und diese haben wiederum einen Zeiger auf ihre gemeinsame Basisklasse „Angestellter":



Das Schöne an virtuellen Basisklassen ist ja, daß man sich eigentlich keine Gedanken darüber machen muß, wie das nun funktioniert - man kann ganz einfach darauf vertrauen, daß es klappt und jede virtuelle Basisklasse in jedem Objekt genau einmal existiert. Natürlich haben Sie keine Gewähr, daß der Compiler die Klassen in der oben dargestellten Reihenfolge anordnet. Die virtuellen Basis-

klassen liegen hinter dem nicht-virtuellen Teil der Klasse, ihre Reihenfolge untereinander ist aber nicht festgelegt.

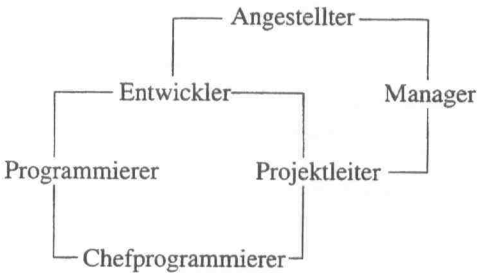
Eine Basisklasse kann innerhalb eines Ableitungsbaums gleichzeitig virtuell und nicht-virtuell sein, z. B. wenn wir `"Programmierer"` nicht-virtuell von `"Entwickler"` ableiten und anschließend daraus einen Chef-Programmierer zusammenbasteln:



bzw. im Quelltext:

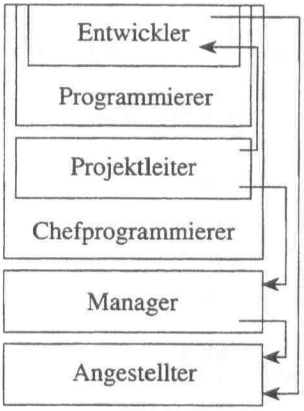
```
class Projektleiter: virtual public Entwickler,
            virtual public Manager ...

class Programmierer: public Entwickler ...

class Chefprogrammierer : public Projektleiter,
              public Programmierer ...
```

Ich habe im Ableitungsbaum absichtlich zwei Fragezeichen gelassen und die Klassendefinitionen nicht endgültig aneinander gesetzt, denn erst muß ich noch erwähnen, daß der Compiler hier merkt, daß die virtuelle Basisklasse `"Entwickler"` aus der Klasse `"Projektleiter"` gleichzeitig nicht-virtuell in `"Programmierer"` vorkommt und deshalb die statische Basisklasse gleichzeitig als virtuelle benutzt:



Na, das ist doch wohl ne Sahne-Klasse, wa? Der Compiler muß jetzt schon ganz schön durch den Klassenbaum navigieren...

Was, Sie wollen ernsthaft wissen, wie das im Speicher aussieht? Na gut, hier ist es:



Wie Sie sehen, handhabt der Compiler Mischformen von virtueller und nicht-virtueller Vererbung durchaus virtuos, aber trotzdem sollte man es vermeiden, damit man nicht selbst den Überblick verliert. Als Faustregel Mehrfach-Vererbung benutzt, grundsätzlich virtuell abzuleiten.

Wenn man nun einen Zeiger auf **`"Chefprogrammierer"`** hat und ihn in einen Zeiger auf **`"Angestellter"`** umwandeln will, genügt es natürlich nicht, wenn der Compiler einfach einen Offset addiert. Vielmehr muß er sich auf einen von mehreren möglichen Pfaden an Zeigern entlanghangeln, um ans Ziel zu gelangen. Dummerweise führt hier kein Pfeil von **`"Angestellter"`** zurück nach **`"Chefprogrammierer"`**, und durch scharfes Nachdenken wird man einsehen, daß es auch nicht sinnvoll, ja sogar nahezu unmöglich ist, einen solchen Zeiger zusätzlich einzuführen. Deshalb ist es in C++ grundsätzlich nicht möglich, einen Zeiger auf eine Klasse in einen Zeiger auf eine virtuell abgeleitete Klasse umzuwandeln:

```
class VirtBasis
{ char Member;  };
```

```
class VirtAbgeleitet : virtual public VirtBasis
{ short Noch_n_Member; };

VirtBasis B1;
VirtAbgeleitet A1;

VirtBasis *bp1 = &A1;          // OK

VirtAbgeleitet *ap1 = (VirtAbgeleitet*) bp1; // ERROR, und
                      // zwar trotz Cast!
```

Das klingt vielleicht wie eine gravierende Einschränkung, aber in der Realität wird man so etwas nötigenfalls elegant durch virtuelle Member-Funktionen lösen, denn selbstverständlich kann man die auch bei virtueller Vererbung benutzen.

Apropos virtuelle Funktionen: Bekanntlich hat eine Klasse mit virtuellen Funktionen einen einzigen Zeiger auf eine Tabelle mit den Daten aller virtuellen Funktionen, während für jede virtuelle Basisklasse eine Zeiger darauf direkt in die Klassenstruktur eingesetzt wird. Warum diese beiden unterscheidlichen Lösungen? Könnte man nicht auch die Aufrufsdaten für die Funktionen direkt in die Struktur einsetzen oder umgekehrt auch virtuelle Basisklassen über eine Tabelle verwalten und dann nur noch einen Zeiger auf diese Tabelle in dem Klassenobjekt aufbewahren, vielleicht sogar eine gemeinsame Tabelle für virtuelle Funktionen und virtuelle Basisklassen einführen?

Im Prinzip sind beide Alternativ-Lösungen praktikabel. MaxonC^{++} verwaltet aber virtuelle Funktionen über Tabellen und virtuelle Basisklassen über direkte Zeiger, und das aufgrund der folgenden praxisnahen Abschätzungen:

1. Laufzeit: Ein Funktionsaufruf ist sowieso eine aufwendige Angelegenheit, und so ist es nicht weiter schlimm, wenn bei einer virtuellen Funktion eine zusätzliche Pointeroperation nötig wird. Dagegen erwartet man als Programmierer, daß der Zugriff auf einen Daten-Member schnell erfolgt, und so ist hier die schnellstmögliche Lösung, nämlich die Zeiger auf die Basisklassen direkt in jedem Klassenobjekt unterzubringen, wohl die sinnvollste.

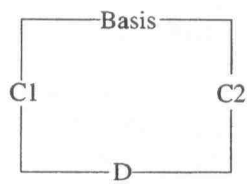
2. Speicherplatz: Eine Klasse hat bei allen denkbaren Anwendungen nur eine sehr begrenzte Anzahl von Basisklassen, selten wird diese Zahl in der Praxis drei oder vier überschreiten. Da jede dieser virtuellen Basisklassen natürlich selbst einige Daten enthält, kann man davon ausgehen, daß eine Klasse mit z. B. vier Basisklassen einen gewissen Umfang besitzt und die vier zusätzlichen Pointer für die Basisklassen nicht weiter ins Gewicht fallen. Andererseits kommt es vor, daß auf einer Klasse, die nur einige wenige kleine Datenmember enthält, etliche virtuelle Funktionen deklariert sind, und im Laufe des Programms zahllose Objekte dieser kleinen Klasse erzeugt werden. Dann wäre es einfach Wahnsinn, in jedes Objekt für jede einzelne Funktion einen Zeiger und einen Offset einzutragen - der Speicherbedarf wäre enorm.

Man beachte, daß die beiden „virtuellen Features" in MaxonC^{++} (und in jedem anderen C^{++} wohl auch) so implementiert sind, daß die Grundregel von C^{++} nicht verletzt wird: Alles ist so zu implementieren, daß weder in der Laufzeit noch im Speicherbedarf ein Overhead im Vergleich zu nichtobjektorientierten Lösungen entsteht.

Es können schon eigenartige Effekte entstehen, wenn zwei Klassen sich eine Basisklasse „teilen". Dazu gehört auch, daß diese Basisklasse (theoretisch, denn praktisch wird man das wohl vermeiden) unterschiedliche Zugriffsrechte hat:

```
class Basis
{ public:
    int Basismember;
};

class C1 : public virtual Basis { };

class C2 : private virtual Basis { };

class D: public C1, public C2 { };
```

Schauen wir uns ganz schnell den Ableitungsbaum dazu an:



Die Frage, die der aufmerksame Leser sich hier stellen wird, lautet: Ist `"Basis"` in `"D"` öffentlich oder privat? Offensichtlich kommt das ganz darauf an, wo man im Baum lang geht, über `"C1"` oder über `"C2"`. Das kann man in einem Programm durch entsprechende Typwandlungen auch explizit tun:

```
void f(D* dp)
{ dp ->Basismember = 42;                // Nanu?
  ((C1*) dp)->Basismember = 43;     // Über C2, also OK
  ((C2*) dp)->Basismember = 43;     // Über C3, also ERROR
}
```

Wie Sie sehen, ist es möglich, von der Klasse `"D"` aus auf den Namen `"Basismember"` zuzugreifen, zumindest mit etwas Aufwand. Deshalb hat man sich entschieden, dort, wo oben „Nanu?" steht, den Zugriff zu erlauben, nach dem Motto, daß es sinnlos ist, Verbote aufzustellen, die man sowieso problemlos umgehen kann.

Ganz allgemein lautet die Regel also: Wenn auf einen Namen auf verschiedenen Wegen mit unterschiedlichen Rechten zugegriffen werden kann, gilt der jeweils geringste Schutz. Ich kann mir aber momentan keine Anwendung vorstellen, bei der man auf die Idee kommen könnte, bei Mehrfach-Vererbung von einer Basisklasse mit unterschiedlichen Zugriffsrechten abzuleiten.

3.3 Konstruktoren und Destruktoren

3.3.1 Initialisierung von Klassen

Lieber Leser, endlich ist Licht am Ende des Tunnels zu sehen: Es fehlt uns nur noch ein wesentliches OO-Konzept von C++, nämlich das der Konstruktoren und Destruktoren.

Als Programmierer steht man eigentlich ständig vor dem Problem, daß ein Datenobjekt initialisiert werden muß, bevor man es benutzen kann. Nun tritt die OO-Programmierung mit dem Anspruch an, daß die Daten des Programms eine gewisse lokale Intelligenz besitzen, und so ist es eine naheliegende Idee, die Initialisierung von Objekten zu automatisieren. In C++ und etlichen anderen Programmiersprachen behandelt man dieses Konzept unter dem Begriff „Konstruktoren".

Ein Konstruktor ist eine Member-Funktion, die automatisch aufgerufen wird, sobald ein Objekt einer Klasse erzeugt wird. Wenn eine Klasse einen solchen Konstruktor besitzt, ist es zwar nicht unmöglich, aber doch ziemlich schwierig, ein nicht initialisiertes Datenobjekt zu erhalten.

Ein Konstruktor kann Parameter haben, und wie jede andere Funktion auch kann man Konstruktoren überladen. Betrachten wir zunächst den einfachsten Fall: Eine Klasse besitzt nur einen Konstruktor, der keine Argumente erwartet. Man kann solche Konstruktoren z. B. benutzen, um alle Member einer Klasse ordentlich mit Null oder einem anderen sinnvollen Wert zu initialsieren. Damit wir aber auch etwas sehen, macht der Konstruktor im folgenden Beispielprogramm auch eine kleine Bildschirmausgabe:

```
#include <stream.h>

class Beispiel
{ public:
     int i;
     char *str;
     Beispiel();    // Das ist ein Konstruktor
};

Beispiel::Beispiel()
{ i = 0;
  str = "Hello, World!";
  cout << str;
}

void main()
{ Beispiel b1;  }
```

Wie Sie sehen, deklariert man eine Konstruktor-Funktion, indem man einer Member-Funktion denselben Namen gibt, den auch die Klasse trägt. Ein Konstruktor darf keinerlei Ergebnistyp haben (nicht einmal `"void"`) und nicht als `"static"` deklariert werden, was auch einleuchtend ist, denn seine Aufgabe ist schließlich, ein konkretes Objekt zu initialisieren. Aus demselben Grund darf er auch nicht als `"const"` deklariert werden. Weil ein Konstruktor normalerweise nur dann aufgerufen wird, wenn ein neues Datenobjekt erzeugt wird, weiß der Compiler stets, mit waelcher Klasse

er es zu tun hat, und so ist es sowohl sinnlos als auch verboten, einen Konstruktor als virtuell zu deklarieren.

Das Hauptprogramm besteht oben ausschließlich aus einer Deklaration einer Variablen des Typs `"Beispiel"`. Der Compiler weiß aber, daß diese Klasse einen Konstruktor besitzt, und erzeugt gleich bei der Deklaration dieser Variablen einen entsprechenden Konstruktoraufruf, durch den `"b1"` initialisiert wird. Außerdem hat der Konstruktor im Beispiel noch einen Seitenefekt in Form einer Ausgabe, so daß hier scheinbar aus dem Nichts (auch bekannt als `"Nil:"`) der Text **„Hello, World"** ausgegeben wird.

Daraus erkennen Sie, daß man Konstruktoren sehr schön dazu mißbrauchen kann, ein Programm durch solche Seiteneffekte vollkommen unlesbar zu machen. Das ist aber natürlich nicht der Sinn der Sache - Konstruktoren sollten ausschließlich zur Initialisierung von Objekten benutzt werden.

Es ist möglich, einen Konstruktor als `"private"` oder `"protected"` zu deklarieren. Dann können nur die entsprechend berechtigten Programmteile Objekte dieser Klasse deklarieren.

Zum Schluß sei noch erwähnt, daß die Scope-Regeln hier etwas komisch, aber durchaus sinnvoll deklariert sind, denn die Deklaration eines Konstruktors, der ja immer so heißt wie seine Klasse, verdeckt keineswegs den Klassennamen:

```
class OK
{ OK *zeiger1;
  OK();          // Konstruktor heißt wie Klasse
  OK *zeiger2;     // OK, ist immer noch Klassenname
};
```

Diese Regelung ist zwar inkonsistent, aber sinnvoll.

3.3.2 Konstruktoren mit Argumenten

Ein Objekt mit sinnvollen Default-Werten initialisieren zu lassen, ist zwar ganz praktisch, aber noch nicht die Erfüllung der letzten Träume des Programmierers. Deshalb bietet C++ auch die Möglichkeit, Konstruktoren mit Argumenten aufzurufen und überdies Konstruktoren zu überladen.

Als Beispiel wählen wir uns hier einmal die komplexen Zahlen, die bekanntlich aus einem Realteil und einem Imaginärteil bestehen. Diese beiden Teile sollten stets mit zwei `"double"`-Werten initialisiert werden:

```
class Complex
{ public:
     double re, im;
     Complex(double, double);
};
```

Jetzt akzeptiert der Compiler es nicht mehr, wenn man ein "Complex"-Objekt „einfach so" deklariert:

```
Complex c1;    // ERROR: No default constructor

Statt dessen haben Sie hier gleich zwei verschiedene Möglichkeiten,
das Objekt zu initialisieren:

Complex c2(1.0, -2.0), c3(0, 0);    // Erste Möglichkeit

Complex c4 = Complex(17.4, 0.42);   // Zweite Möglichkeit
```

Die beiden Schreibweisen sind äquivalent, wobei die erste Alternative an Simula angelehnt ist (was, Sie kennen Simula nicht? Na ja, wahrscheinlich waren Sie damals noch zu jung), während die Syntax mit dem "=" mehr Ähnlichkeit mit einer „normalen" Initialisierung in C hat.

Natürlich werden Konsrtuktoren nicht nur bei Variablendeklarationen aufgerufen, sondern nahezu immer, wenn ein Objekt erzeugt wird, und folglich auch bei "new":

```
Complex *cp = new Complex(17.4, 21.0);
Complex *cpvector = new Complex[100];    // ERROR
```

Das zweite "new" ist nicht erlaubt, weil man bekanntlich Vektoren bei "new" nicht initialisieren kann, die Klasse "Complex" aber eine Initialisierung verlangt.

Konstruktoren lassen sich überladen, wie jede andere Funktion auch:

```
class Complex2
{ public:
     double re, im;
     Complex2(double r, double i)
     {  re = r;
        im = i;
     }
     Complex2(double x)
     { re = x;
        im = 0;
     }
     Complex2()
     { re = 0;
        im = 0;
     }
};

Complex2 d1, d2(3), d3(1,2);

Complex2 d4 = 1.2;

Complex2 d5(d1);
```

Hier wurden die Konstruktoren sogar zusätzlich als `"inline"` definiert, was bei so kleinen Funktionen sinnvoll ist. Natürlich hätte man dasselbe auch durch entsprechende Default Parameter erledigen können, aber Überladen ist nun einmal schöner.

Bei `"d4"` sehen Sie noch eine dritte mögliche Schreibweise: Wenn ein Konstruktor nur ein Argument erwartet, kann man auf diese Weise auch direkt initialisieren. Auf diese Weise erhält man gewissermaßen eine selbstdefinierte Typkonvertierungsregel.

Bei `"d5"` wird angedeutet, daß jede Klasse automatisch einen Konstruktor besitzt, nämlich den Kopier-Konstrutor, der ein neues Objekt erzeugt, indem er ein bereits existierendes kopiert. Wenn der Programmierer nicht selbst einen Kopier-Konstruktor der Form

```
class X
{ // ...
  X(const X&);      // Alternativ auch ohne "const"
};
```

deklariert, macht das der Compiler. Das folgende ist kein Kopier-Konstruktor,

```
class Y
{ // ...
  Y(Y);
 };
```

weil für den Aufruf dieser Funktion das Argument auf den Stack kopiert werden müßte, und dazu wäre erneut der Aufruf eines Kopier-Konstuktors notwendig.

3.3.3 Initialisierung von Basisklassen und Membern

Jetzt stellt sich natürlich die Frage, wie ein Konstruktor für abgeleitete Klassen auszusehen hat, schließlich müssen ja auch Basisklassen sauber initialisiert werden. Deshalb hat man für Konstruktor-Definitionen eine besondere Syntax eingeführt. Vor den eigentlichen Anweisungsteil der Funktion sind die Initialisierungen der Basen zu setzen, z. B. so:

```
class A
{ // usw.
  public:
    A(int);
};

class B
{ // usw.
  public:
    B(int, int);
};

class C: public A, private B
{ // Nochwas...
  C(int, int, int);
};
```

```
C::C(int a, int b, int c) : A(a), B(b, c)
{ // Sonstige Initialisierungen...
}
```

Im Beispiel hat die Klasse "C" zwei Basisklassen, die beide keinen Default-Konstruktor haben. Deshalb verlangt der Compiler hier, daß "C" einen Konstruktor haben muß, und meldet sonst einen Fehler. Bei der eigentlichen Definition des Konstruktors sind dann die Initialisierungen der Basisklassen entsprechend vorzunehmen. Auch hier wacht der Compiler pingelig darüber, daß keine Basisklasse uninitialisiert bleibt. Wenn eine Basisklasse aber einen Default-Konstruktor hat, kann man ihre Initialisierung weglassen, und jener Konstruktor wird automatisch aufgerufen.

Mit derselben Syntax kann man auch Member initialisieren, was vor allem bei Membern von "const"- oder Referenztypen nötig ist, denn die können nicht durch eine gewöhnliche Wertzuweisung initialisiert werden. Das folgende Listing illustriert dieses Problem und seine Lösung:

```
class Falsch
{ int &ir;
  const int ic;
};          // ERROR: "ir" und "ic" nicht initialisiert

class AuchFalsch
{ int &ir;
  const int ic;
  AuchFalsch(int&);
};                    // So weit OK

AuchFalsch::AuchFalsch(int &ref)
{ ir = ref;             // Keine wirkliche Initialisierung!
  ic = ref;             // Verboten, da "const"
}

class Richtig
{ int &ir;
  const int ic;
  Richtig(int&)
};                    // OK, hat Konstruktor

Richtig::Richtig(int &ref) : ir(ref), ic(ref)
{ cout << ref;    }
```

Bei solchen Konstruktor-Initialisierungen ist die Reihenfolge von Basisklassen und Membern beliebig. Es versteht sich, daß auch Member mit Konstruktoren auf diese Weise initialisiert werden können - oder sogar müssen, wenn kein Default Konstruktor vorhanden ist:

```
class BuM      // "Basis und Member"
{ int i;
  public:
    BuM(int);
};
```

```
class Abgel : public BuM
{ BuM bumm;
  public:
    Abgel(int);    // Klasse MUSS Konstruktor haben!
};
// Der Konstruktor für "Abgel" konstruiert eine Basisklasse und
// einen Member, beide vom gleichen Typ "BuM":

Abgel::Abgel(int i) : BuM(i+1), bumm(i+2)
{ // Sonstige Initialisierungen ...
}
```

Etwas komisch wird die Angelegenheit bei Mehrfach-Vererbung mit virtuellen Basisklassen:

```
class Basis
{ int i;
  public:
    Basis(int);
};

class Ab1 : public virtual Basis
{ int j;
  public:
    Ab1(int, int);
};

Ab1::Ab1 (int p1, int p2) : Basis(p1)
{ j = p2;  }

class Ab2 : public virtual Basis
{ int k;
  public:
    Ab2(int, int);
};

Ab2::Ab2 (int p1, int p2) : Basis(p1)
{ k = p2;}

class Top: public Ab1, public Ab2
{ public:
    Top(int, int, int);
};

Top::Top (int p, int q1, int q2) : Ab1(p, q1), Ab2 (p, q2)
{ }
```

Die Klasse `"Top"` hat die Basisklassen `"Ab1"` und `"Ab2"`, die sich eine gemeinsame Basisklasse `"Basis"` teilen. Alle Klassen initialisieren über Konstruktoren ihre Basisklassen, was eigentlich zur Folge haben würde, daß `"Basis"` gleich zweimal initialisiert würde, was nicht nur Zeitverschwendung wäre, sondern auch zu haarsträubenden Inkonsistenzen führen würde. Stellen Sie sich beispielsweise einmal vor, der Konstruktor von `"Basis"` würde jeweils eine statische Variable inkre-

mentieren, damit man immer weiß, wie viele Objekte der Klasse `"Basis"` es gibt. Wenn dann ein Objekt der Klasse `"Top"`, das nur ein `"Basis"`-Objekt enthält, erzeugt wird, würde dieser Zähler gleich doppelt inkrementiert, was ganz sicher nicht im Sinne des Erfinders ist.

Keine Sorge, genau das passiert nicht. Es wird automatisch geregelt, daß virtuelle Basisklassen nur genau einmal konstruiert werden, und zwar von einer bestimmten abgeleiteten Klasse. Die Regeln, welche Klasse das jeweils ist, sind ziemlich kompliziert und zum Teil implementationsabhängig, deshalb sollten Sie sich darum am besten gar nicht kümmern. Im obigen Beispiel ist es sowieso egal, ob `"Basis"` von `"Ab1"` oder `"Ab2"` aus initialisiert wird, denn in beiden Fällen wird der Konstruktor mit demselben Argument aufgerufen.

Implementiert wird dieses Feature übrigens, indem in Klassen mit einer virtuellen Basisklassen nicht nur der entsprechende Zeiger, sondern auch ein Flag, ob diese Basisklasse zu initialisieren ist, eingetragen wird. Mit demselben Mechanismus wird auch dafür gesorgt, daß eine Klasse genau einmal destruiert wird.

In Konstruktoren und Destruktoren werden Member-Funktionen nicht virtuell aufgerufen:

```
class Base
{ public:
    virtual void f();
};

class Derived : public Base
{ int member;
  public:
    Derived();
};

Derived::Derived()
{ member = 26731;
  f();          // Nicht virtuell!!
}
```

Warum das? Nun, es könnte ja sein, daß von `"Derived"` weiter abgeleitet wird, etwa so:

```
class D2 : public Derived
{ public:
    int &ir;    // Braucht Initialisierung
    void f();
    D2();
};

void D2::f()
{ ir = 42; }

D2::D2() : Derived(), ir(member)
{ // usw.  }
```

Basisklassen werden grundsätzlich vor abgeleiteten Klassen konstruiert. Wäre der Aufruf von `"f"` im Konstruktor von `"Derived"` virtuell, so würde hier `"D2::f"` aufgerufen, wo die Referenz `"ir"` benutzt wird, bevor sie initialisiert wird, und das darf in C++ einfach nicht sein.

3.3.4 Besondere Konstruktoren

3.3.4.1 Default-Konstruktoren

Konstruktoren können beliebige und beliebig viele Argumente haben, ganz wie es für die Initialisierung der Klasse nötig ist. Es gibt aber zwei Arten von Konstruktoren, die aus dem Rahmen fallen: die argumentlosen Default-Konstruktoren und die Kopier-Konstruktoren.

Ein Default-Konstruktor ist ein solcher, der ohne Argumente aufgerufen werden kann. Dafür gibt es mehr Möglichkeiten, als Sie vielleicht denken:

```
class C1
{ public:
    C1();      // Einfachster Fall
};

class C2
{ public:
    C2(...);
};

class C3
{ public:
  C3(int i = 17, double d = 4.0);
};

C1 c1;         // OK
C2 c2;         // OK
C3 c3;         // OK
```

Objekte dieser drei Klassen darf man ohne Argumente bzw. Initialisierung aufrufen, denn der Compiler kann jeweils einen Konstruktor ohne Argumente aufrufen. Als kleine Inkonsistenz darf man auch Klassen, die überhaupt keinen Konstruktor besitzen, uninitialisiert erzeugen, dann findet entsprechend keinerlei Initialisierung statt.

Vektoren von Klassen lassen sich, sofern man nicht den Default-Konstruktor benutzen will oder kann, auch über Elementlisten konstruieren:

```
class C
{ short nr, x;
  public:
    C(int);   // Kein Default-Konstruktor
};

C cv[5] = { 2, 4, 6, 8, 10 };     // Konstruktor-Aufrufe!
```

Hier wird für jedes Vektorelement ein Konstruktoraufruf mit der entsprechenden Nummer generiert. Diese Syntax kann aber nur eingeschränkt werden, z. B. nicht zur Initialisierung von mit `"new"` erzeugten Vektoren oder für Vektor-Member:

```
class D
{ C vectormember[3];
  D();          // Konstruktor nötig
}

D::D() : vectormember ( { 0, 8, 15 } )   // ERROR!
{ }
```

Formal wird dieser Fehler damit begründet, daß eine Elementliste im Sinne von C kein „Ausdruck" ist. Wenn Sie sich fragen, wie man obiges Programm richtig schreiben soll, so muß ich leider sagen, daß das unmöglich ist - Member dürfen nur dann Vektoren einer Klasse sein, wenn eben jene Klasse einen Default-Konstruktor besitzt.

3.3.4.2 Kopier-Konstruktoren

Jede Klasse besitzt automatisch einen Kopier-Konstruktor, denn wenn man nicht selbst einen deklariert, tut es der Compiler automatisch.

Normalerweise wird ein Kopier-Konstruktor folgendermaßen deklariert (siehe auch 3.3.2):

```
class C
{ // ...
  C(const C&);
};
```

„Normalerweise" deshalb, weil es in der Regel nicht nötig sein wird, ein Objekt zu verändern, wenn man es bloß in ein anderes kopieren will. Falls doch, ist

```
class Seltsam
{ // ...
  Seltsam(Seltsam &);
};
```

auch ein gültiger Kopier-Konstruktor. Grundsätzlich sollte man aber Referenzen auf Objekte, die man nicht verändern will, als `"const"` deklarieren.

Wenn der Compiler selbst einen Copy-Konstruktor definiert, sieht der so aus, daß alle Member und Basisklassen einzeln kopiert werden. Wenn ein Member oder eine Basisklasse aber einen eigenen Kopier-Konstruktor besitzt, wird natürlich dieser benutzt. Ein vom Compiler generierter Konstruktor ist stets `"public"`.

Kopier-Konstruktoren werden immer dann aufgerufen, wenn ein Objekt initialisiert wird, nicht jedoch bei einer gewöhnlichen Wertzuweisung:

```
#include <stream.h>

class C
```

```
{ public:
    int i;
    C(const C&);
    C(int);
};

C::C(const C &c2)    // Copy-Konstruktor
{ i = c2.i;
  cout « "Copy-Konstruktor ausgeführt.\n";
};

C::C(int i1) : i(i1) { }   // Gewöhnlicher Konstruktor

// Eine Funktion, die ein Objekt als Parameter
// erwartet und zurückgibt:
C f(C carg)
{ C result(carg);  // ** 1
  result.i += 42;
  return result;   // ** 2
}

void main()
{ C c1 = 17, c2 = c1;   // ** 3

  c2 = c1;     // KEIN Copy-Konstruktor-Aufruf

  c2 = f(c1); // ** 4
}
```

In diesem Programm wird der Kopier-Konstruktor genau viermal aufgerufen, nämlich überall da, wo `"***"` steht:

1. In `"f"` wird das Objekt `"result"` deklariert und mit `"carg"` initialisiert. Unabhängig davon, ob hier die Schreibweise `"C x(y)"` oder `"C x = y"` benutzt wird, ist dies der „klassische" Fall für den Aufruf eines Copy-Konstruktors.

2. Bei `"return"` wird ein Klassenobjekt zurückgegeben. Implementatorisch reicht es natürlich nicht, einen Zeiger auf das lokale Objekt `"result"` zurückzugeben, denn nach Beendigung der Funktion existiert dieses Objekt ja nicht mehr. Statt dessen muß der Compiler irgendwo ein namenloses temporäres Objekt einrichten, in das das Funktionsergebnis kopiert wird, und weil dieses Phantom-Objekt mit dem Funktionsergebnis initialisiert wird, wird folgerichtig auch bei `"return"` generell ein Konstruktor aufgerufen.

3. In `"main"` wird das Objekt `"c2"` mit dem gleichartigen Objekt `"c1"` initialisiert, wobei diesmal die Schreibweise mit dem `"="` benutzt wird. Auch dies ist ein ganz klarer Fall für den Copy-Konstruktor. In der folgenden Zeile findet dagegen keine Initialisierung, sondern eine ganz gewöhnliche Wertzuweisung statt und deshalb wird dort auch kein Konstruktor aufgerufen. Man kann auch diese normale Wertzuweisung verändern, indem man den Operator `"="` überlädt, aber das gehört in ein anderes Kapitel.

4. Zum Schluß wird die Funktion `"f"` aufgerufen, die als Argument ein Objekt der Klasse `"C"` erwartet (und nicht etwa eine Referenz darauf). Also wird für diesen Funktionsaufruf ein neues Objekt erzeugt, das für die Funktion später `"carg"` heißen wird, und natürlich muß dieses neue Objekt konstruiert werden. Also wird `"carg"` mit dem Copy-Konstruktor aus `"c1"` erzeugt. Unter (2.) wurde bereits angedeutet, daß das Funktionsergebnis bei `"return"` in ein temporäres Objekt kopiert bzw. konstruiert wird. Nachher wird dieses temporäre Objekt nach `"c2"` kopiert, aber das ist wieder eine ganz normale Wertzuweisung und führt deshalb zu keinem weiteren Aufruf des Kopierkonstruktors.

An dieser Stelle möchte ich noch kurz darstellen, wie die Rückgabe eines Klassenobjekts in MaxonC++ intern realisiert wird. Bei numerischen oder Pointerdaten wird das Ergebnis in einem bestimmten Prozessorregister, nämlich `"d0"`, zurückgegeben, aber ein Klassenobjekt wird meist nicht in einem Register Platz haben. Deshalb erhält eine Funktion, die eine Struktur (bzw. ein Objekt) zurückgeben soll, als unsichtbaren Parameter einen Zeiger auf den Speicherbereich, in den das Ergebnis zu kopieren ist, wobei aber stets der Kopier-Konstruktor verwendet wird. Der Compiler erzeugt bei Bedarf ein temporäres Objekt, in dem das Ergebnis konstruiert wird, und generiert dann Code, der die Daten aus dem temporären Objekt in das eigentliche Zielobjekt kopiert, wobei auch ein überladener `"="`-Operator Beachtung findet.

Auf den ersten Blick erscheint dieses Hin- und Herkopieren vielleicht ziemlich umständlich, denn man könnte doch auch das Ergebnis direkt im Zielobjekt konstruieren. Aber in C++ wird streng zwischen Initialisierungen und Wertzuweisungen unterschieden, und ein Ausdruck `"y = f(x)"` ist nun einmal eine Wertzuweisung und keine Initialisierung.

3.3.5 Destruktoren

Konstruktoren stellen ein leistungsfähiges Konzept dar, mit dem man leicht Objekte realisieren kann, die sich automatisch initialisieren und schon bei ihrer Erzeugung komplexe Aufgaben erledigen. Nun bleibt noch das Problem, daß man oft „Aufräumen" muß, sobald ein Objekt zu existieren aufhört. Aber auch das ist in einer Programmiersprache wie C++ kein Problem, denn hier gibt es das Destruktor-Konzept.

Ein Destruktor ist eine Funktion, die der Compiler automatisch immer dann aufruft, wenn ein Objekt der jeweiligen Klasse gelöscht wird, d. h. für statische Variablen am Programmende für Variablen im automatischen Speicher wenn der Block, in dem sie deklariert sind, beendet wird für durch `"new"` erzeugte Objekte beim Aufruf von `"delete"`.

Man deklariert einen Destruktor als Member-Funktion, die wie ein Konstruktor (also so wie die Klasse) heißt, nur daß man dem Namen ein `"~"` voranstellt:

```
#include <stream.h>

class Test
{ int nr;
  public:
    Test(int);          // Konstruktor
```

```
    ~Test();            // Destruktor
};

Test::Test(int i) : nr(i)
{ cout « "Objekt " « nr « " erzeugt.\n";
}

Test::~Test()
{ cout « "Objekt " « nr « " gelöscht.\n";
}

void main()
{ Test t1(42), t2(26731);
  cout « "Daten sind initialisiert.\n";
}
```

Das Programm erzeugt folgende Ausgabe:

Objekt 42 erzeugt.

Objekt 26731 erzeugt.

Daten sind initialisiert.

Objekt 26731 gelöscht.

Objekt 42 gelöscht.

Am Ende der Funktion `"main"` endet auch die Existenz der Objekte `"t1"` und `"t2"`, und deshalb erzeugt der Compiler hier automatisch die entsprechenden Destruktoraufrufe - wie man sieht, in der umgekehrten Reihenfolge, in der die Objekte erzeugt wurden. Hier ist ein anderes Beispiel: Ein objekt-orientiertes „Hello World"-Programm:

```
#include <stream.h>

class Hallo
{ public:
    Hallo();
    ~Hallo();
};

Hallo::Hallo()
{ cout « "Hello ";
}

Hallo::~Hallo()
{ cout « "World!";
}

Hallo H;

void main()
{ }
```

Man beachte vor allem, daß hier das alles außerhalb des Hauptprogramms geschieht, nämlich in den Konstruktoren und Destruktoren, die für die globale Variable `"H"` aufgerufen werden. Solche statischen Variablen mit Filescope werden vor Ausführung von `"main"` konstruiert und am Programmende destruiert. Im Prinzip kann man auf diese Weise Programme schreiben, die ganz ohne Hauptprogramm auskommen, aber der Linker will trotzdem `"main"` haben.

Ein Destruktor darf keine Parameter haben - klar, denn der Compiler muß ihn ja selbständig aufrufen und kann sich keine Argumente aus dem Finger saugen. Folglich kann man einen Destruktor nicht überladen, und außerdem darf er kein Ergebnis zurückgeben (noch nicht einmal `"void"`). Genau wie ein Konstruktor darf ein Destruktor nicht als `"static"` deklariert werden. Es macht nichts, wenn er `"private"` oder `"protected"` ist, der Compiler darf ihn trotzdem von jedem Programmteil aus aufrufen. Bei Destruktoren sagt der Zugriffsmodus nur aus, ob der Programmierer ihn explizit aufrufen darf.

Ein Destruktor wird auch bei `"delete"` aufgerufen. Im Gegensatz zu `"new"` ist bei `"delete"` die genaue Klasse des dynamisch erzeugten Objekts nicht zur Compilezeit bekannt, und deshalb ist es erlaubt und vor allem auch sinnvoll, einen Destruktor als virtuell zu deklarieren:

```
#include <stream.h>

class Basis
{ virtual ~Basis()
   { cout « "Basis destruiert.\n";
   }
};

class Klasse: public Basis
{ virtual ~Klasse()
   { cout « "Klasse destruiert.\n";
   }
};

void main()
{ Basis *bp = new Klasse;

   delete bp;
}
```

Der Destruktor von `"Klasse"` überdeckt virtuell den von `"Basis"`, und so wird bei `"delete bp"` der korrekte Destruktor aufgerufen. Die Ausgabe des Programms lautet also:

```
Klasse destruiert.   Basis destruiert.
```

Daraus lernen Sie auch unschwer, daß Basisklassen (und Member) nach Beendigung eines Destruktors automatisch ebenfalls destruiert werden, so daß der Destruktor einer abgeleiteten Klasse sich darum nicht explizit zu kümmern braucht. Destruktoren funktionieren also völlig analog zu Konstruktoren, und deshalb wird es Sie auch nicht überraschen, daß

- Member-Funktionen von Destruktoren aus nicht virtuell aufgerufen werden (siehe 3.3.3),

- beim Löschen von Vektoren von Klassen der Destruktor für jedes einzelne Vektorelement aufgerufen wird, und
- virtuelle Basisklassen bei Mehrfach-Vererbung nur genau einmal destruiert werden (hier werden dieselben Flags wie beim Konstruieren verwendet, siehe 3.3.3).

Ansonsten ist noch zu erwähnen, daß der Compiler Destruktoren (genau wie Konstruktoren) ziemlich clever handhabt:

```
class C
{ public:
    C();
    ~C();
};

void main()
{ goto L1;
  C c1;  // ERROR: Jump past initialisation of "c1"
         // (d. h. es ist nicht mehr klar, ob "c1" später
L1:
         // destruiert werden soll)

  if(1)
  { C c2;
    goto L2;  // Vor "goto" wird "c2" automatisch destruiert
    C c3;     // Destruktor-Aufrufe für "c2" und "c3"
  }

  L2:  C c4;
  if (0)
    goto L2;  // Bei "goto": Destruktor-Aufruf für "c4"

  // Hier werden "c4" und "c1" destruiert
}
```

Durch solche Späßchen garantiert der Compiler, daß zu jedem Konstruktor- genau ein Destruktoraufruf erfolgt, ganz so wie man es erwartet.

3.4 Beispielprogramm:

Telefon-Listen, diesmal aber objekt-orientiert

Am Ende von Kapitel 2 habe ich Ihnen als Beispiel ein kleines Telefonbuch-Programm vorgestellt, und am Anfang dieses Kapitels versprach ich Ihnen, daß man so etwas objekt-orientiert viel eleganter lösen kann. Jetzt, da es sowieso wieder einmal Zeit wird, ein Demoprogramm zu bringen, will ich mein Versprechen einlösen: Es folgt eine neue Version des Telefonbuch-Programms und zwar unter Benutzung der meisten objektorientierten Features, die bis jetzt vorgestellt wurden.

Ich möchte Sie aber lieber gleich warnen: Das Programm wird dadurch keineswegs einfacher! Ich will hier nämlich zugleich ein Beispiel für abstrakte Datentypen bringen, d. h. zunächst eine ganz all-

gemeine Listenstruktur realisieren und sie dann durch Vererbung zu einem Telefonbuch spezialisieren. Also, an's Werk:

```
// Objektorientiertes Telefonlisten-Programm

// Dieselben Header-Files wie in der ersten Version:

#include <stream.h>
#include <string.h>
```

Die wichtigste Idee in unserem Programm ist, zuerst zwei Klassen **"Listenelement"** und **"Liste"** zu implementieren, die eine doppelt verkettete, sortierte Listenstruktur wie bei der ersten Version realisieren, aber noch nicht festlegen, welche Daten in dieser Liste gespeichert werden sollen. Wir werden davon die beiden Klassen **"Telefonbuch"** und **"Telefonbucheintrag"** ableiten und so ziemlich einfach aus der allgemeinen Listenstruktur ein Telefonbuch machen. Ebenso gut könnte man sich aber auch eine nach lateinischen Bezeichnungen sortierte Liste von Würmern implementieren, wobei wiederum jedes Wurm-Objekt als Member eine chronologisch geordnete Liste von Publikationen über ihn enthält, und zu jede Publikation eine Liste der Autoren existiert, oder... Der Phantasie sind keine Grenzen gesetzt, und vielleicht werden Sie auch einmal in einem Ihrer eigenen Programme auf die hier dargestellte abstrakte Listenstruktur zurückgreifen.

Als erstes deklarieren wir die Klasse **"Listenelement"**, die irgendein Element in irgendeiner sortierten, doppelt verketteten Liste darstellen soll. Welche Eigenschaften soll diese Klasse besitzen?

Zunächst einmal wären da die Zeiger **"next"** und **"prev"**, genau wie im alten Programm. Daran sollte der unbedarfte „Benutzer" der Klasse nicht herumspielen, und deshalb deklarieren wir sie (und einige andere Bezeichenr auch) als **"private"**. Dafür müssen wir, wie wir später noch sehen werden, die Klasse **"Liste"** als **"friend"** deklarieren, denn von dort aus muß auf diese Zeiger zugegriffen werden.

Da die Liste später sortiert sein soll, brauchen wir eine Vergleichsfunktion. Da wir aber noch nichts, aber auch gar nichts über die spätere Liste wissen, deklarieren wir diese Funktion (namens **"compare"**) als virtuell und zwar als voll virtuell, denn zwei ganz allgemeine Listenelemente zu vergleichen, ist sicher nicht sinnvoll. Somit ist die Klasse **"Listenelement"** auch schon abstrakt, und das ist ja auch so gedacht.

Aus demselben Grund deklarieren wir die Member-Funktion **"ausgabe"**, die die Daten eines Listenelements ausgeben soll, als voll virtuell. Außerdem führen wir der Vollständigkeit halber einen virtuellen Destruktor ein, denn es kann ja sein, daß später eine Klasse mit einem Destruktor von **"Listenelement"** abgeleitet wird, und so gewährleisten wir in weiser Voraussicht, daß auch stets der richtige Destruktor aufgerufen wird. Wir deklarieremn ihn aber nicht als voll virtuell, sondern definieren ihn als Leerfunktion, so daß man in abgeleiteten Klassen einen Destruktor deklarieren kann, aber nicht muß.

Als kleines Gimmick stellen wir allen abgeleiteten Klassen die Zeiger "next" und "prev" zur Verfügung, und zwar in Form der Funktionen "getnext" und "getprev", die als "protected" deklariert werden. So können abgeleitetet Klassen kontrolliert lesend auf diese Zeiger zugreifen. Wir werden dieses Feature in diesem Programm nicht benutzen.

Wenn wir dann noch bedenkan, daß man im Zweifalsfall alles als "const" deklarieren sollte, um möglichst universell verwendbare Funktionen zu erhalten, sieht die Klassendefinition folgendermaßen aus:

```
class Listenelement
{ private: Listenelement *next, *prev;
  // Vorgänger und Nachfolger

  virtual int compare(const Listenelement*) const = 0;
  // Vergleicht Objekt "*this" mit einem
  // anderen Listenelement
  // und gibt -1, 0 oder +1 zurück (wie bei "strcmp")

  virtual void ausgabe() const = 0;
  // Ein Listenelement ausgeben

  virtual ~Listenelement()  { }  // Virtuieller Destruktor -
  // hier nur ein Dummy, kann aber
  // in abgeleiteten Klassen überdeckt werden

  protected:  // Kleine Extras: abgeleitete Klassen dürfen
              // "next" und "prev"  auslesen:
    Listenelement *getnext() const
      { return next;  }
    Listenelement *getprev() const { return prev;   }

    friend class Liste;  // Diese später deklarierte Klasse
// braucht Zugriff auf private Member
};
```

Da hier alle Member-Funktionen entwerder voll virtuell sind (und deshalb nicht definiert werden müssen) oder direkt als "inline" in der Klassendefinition definiert wurden, sind wir mit der Klasse "Listenelement" auch schon fertig und können den abstrakten Datentyp "Liste" entwerfen, der wieder eine ganz allgemeine Listenstruktur darstellen soll und zunächst nichts mit einem Telefonbuch zu tun hat.

Eine "Liste" besitzt wie bisher je einen Zeiger auf das erste und das letzte Listenelement. Auch diese Zeiger sind Privatangelegenheit der Klasse und werden dementsprechend deklariert.

Als nächstes brauchen wir eine Funktion, die ein neues Element an geeigneter Stelle in die Liste einhängt. Die Position kann sie mit der Funktion "compare" aus "Listenelement" herausfinden, aber da wir sonst nichts über den genauen Charakter der Liste wissen, können wir in dieser Struktur kein neues Listenelement erzeugen, sondern brauchen einen Zeiger auf ein fertig alloziertes und initialisiertes Element. Nun gilt es zu verhindern, daß jemand ein von "Listenelement" abgelei-

tetes Objekt, das einen Fadenwurm repräsentiert, in ein Telefonbuch einfügt - das geht, wenn sowohl die Klasse `"Wurn"` als auch `"Telefonbucheintrag"` von `"Listenelement"` abgeleitet sind. Deshalb deklarieren wir die Funktion `"insert"` als `"protected"`, d. h. nur die von `"Liste"` abgeleitete Klasse (z. B. `"Telefonbuch"`) darf neue Elemente in die Liste einfügen.

Eine Liste braucht einen Konstruktor, der die Zeiger `"anfang"` und `"ende"` initialisiert (früher hatten wir dafür ja die Funktion `"Init"`, die explizit aufgerufen werden mußte), und einen Destruktor, der alle Elemente der aus Liste löscht. Außerdem deklarieren wir die Member-Funktion `"ausgabe"`, die alle Elemente der Liste (z. B. das gesamte Telefonbuch) ausgeben soll, und Funktionen zum Suchen und Löschen von Listenelementen. Da wir immer noch nicht wissen, mit was für einer Liste wir es später zu tun haben werden, braucht die Suchfunktion einen Zeiger auf ein `"Muster"`-Listenelement, zu dem sie dann mit der `"compare"`-Funktion eine Entsprechung in der Liste suchen kann.

Damit kommen wir zu folgender Klassendefinition:

```
class Liste
{  private:
     Listenelement *anfang, *ende;  // Erstes und letztes
     // Element Der Liste
   protected:
     void insert(Listenelement *neu);
     // Sucht eine geeignete Position für das
     // Element "*neu" und
     // hängt es in die Liste ein

   public:
     Liste();  // Konstruktor zur Initialisierung
     ~Liste();
     // Destruktor (löscht alle
     // Listenelemente)
     void ausgabe();  // Gibt die ganze Liste aus

   Listenelement *suchen(Listenelement *);
   // Sucht ein Element, das dem
   // Argument-Element "entspricht"
   // (was immer das in einer konkreten Liste heißen mag -
   // jedenfalls ein Element, bei dem "compare"
   // Null liefert)
   // oder Null, wenn kein solches Element existiert.

     void loeschen(Listenelement*);
   // Hängt ein Element aus der Liste aus und löscht es
   // mit "delete"
};
```

Aus didaktischen Gründen implementieren wir jetzt gleich alle Member-Funktionen von `"Liste"`: Zunächst der Konstruktor, der der Funktion `"Init"` in der alten Programmversion zum Verwechseln ähnlich sieht:

```
Liste::Liste()
{ anfang = 0;
  ende = 0;
}
```

Auch den Destruktor kennen Sie noch, auch wenn er früher eine normale Funktion war und "AllesLoeschen" hieß:

```
Liste::~Liste()
{ Listenelement *p = anfang;

  while(p)
  { Listenelement *hilf = p;
    p=p->next;
    delete hilf;
  }
}
```

Die Member-Funktion "ausgabe" muß hier nichts anderes tun, als für alle Listenelemente deren Funktion "ausgabe" aufzurufen:

```
void Liste::ausgabe()
{ for(Listenelement *l = anfang; l; l = l->next)
    l->ausgabe();
}
```

Dann wäre da noch die Member-Funktion "suchen", die aus einer Liste das erste Element herausucht, das im Sinne der Funktion "compare" mit dem Vergleichselement übereinstimmt:

```
Listenelement *Liste::suchen(Listenelement *e1)
{ for(Listenelement *e2 = anfang;
    e2 && e2->compare(e1);
    e2 = e2->next);
  return e2;
}
```

Die Funktion "insert" ist zwar etwas komplizierter, aber das meiste kennen Sie ja schon aus dem ersten Telefonbuchprogramm. Im Gegensatz zur alten Funktion "Einfueg" erzeugt sie kein neues Element (wir befinden uns schließlich auf einer Ebene, wo wir noch überhaupt nicht wissen, welche Art von Daten in der späteren Liste abgespeichert werden sollen), sondern hängt nur ein bereits existierendes Element in die Liste ein. Die Position ermittelt sie anhand der Sortierfunktion "compare" aus der Klasse "Listenelement":

```
void Liste::insert(Listenelement *neu)
{ // Position suchen:
  Listenelement *pos = anfang;
  while (pos && neu->compare(pos) > 0)
  pos = pos->next;

  // Element an Position "pos" in Liste einfügen
  // (prinzipiell genau wie in der ersten Programmversion)
```

```
  if (pos == 0)  // Sonderfall: Anfügen an das Ende der Liste
  { Listenelement *alt = ende;  // altes Listenende

    if (alt != 0) // Allgemeiner Unter-Fall: wirklich anhängen
    { alt->next = neu;
      neu->prev = alt;
      neu->next = 0;
      ende = neu;
    }
    else // Spezieller Spezialfall: Liste ist noch leer
    { neu->next = 0;
      neu->prev = 0;
      anfang = neu;
      ende = neu;
    }
  }
  else
  if (pos->prev == 0)  // Sonderfall: Neues Element
    // an Listenanfang hängen
    { pos->prev = neu;
      neu->next = pos;
      neu->prev = 0;
      anfang = neu;
    }
    else // Allgemeiner Fall: Element in Liste einhängen
    { Listenelement *vor = pos->prev;
      // "neu" zwischen "vor" und "pos" einhängen:
      vor->next = neu;
      neu->prev = vor;
      neu->next = pos;
      pos->prev = neu;
    }
  }
```

Und weiter geht's - mit einer Funktion, die ein Element aus einer Liste aushängt und dann mit `"delete"` vernichtet. Auch diese Funktion leitet sich direkt aus der alten Programmversion ab, und zwar von der Funktion `"Loeschen"`:

```
void Liste::loeschen(Listenelement *E)
  // Löscht das Element, auf das "E" zeigt, aus der Liste
{ // Vorgänger und Nachfolger des zu löschenden Elements
  Listenelement *vor = E->prev, *nach = E->next;

  if (vor)
    // Element hat Vorgänger: Dann dessen Nachfolger umsetzen
    vor->next = nach;
  else // Kein Vorgänger: Dann Listenanfang aktualisieren
    anfang = nach;

  if (nach)
    // Element hat Nachfolger: Dann dessen Vorgänger verändern
```

```
    nach->prev = vor;
  else      // Kein Nachfolger, also Listenende anpassen
    ende = vor;

  delete(E);
}
```

Wow, das war auch schon fast alles - jedenfalls haben wir damit einen leistungsfähigen abstrakten Datentypen implementiert. Jatzt müssen wir nur noch einen ganz konkreten Datentypen, nämlich das bekannte und beliebte Telefonbuch, daraus ableiten.

Zunächst entwickeln wir den Typen **"Eintrag"** aus **"Listenelement"**, d. h. wir leiten eine neue Klasse ab. Später werden wir sehen, daß es sinnvoll ist, wenn jeder weiß, daß ein **"Eintrag"** von einem **"Listenelement"** abgeleitet ist, deshalb erfolgt die Vererbung hier über **"public"**.

Ein Telefonbucheintrag besteht nach wie vor aus einem Namen und einer Nummer, und auch diese sind **"public"**, damit jeder darauf hemmungslos zugreifen kann. Ferner brauchen wir noch einen Konstruktor, denn durch scharfes Nachdenken stellen wir fest, daß es nirgendwo sinnvoll ist, ein uninitialisiertes **"Eintrag"**-Objekt zu haben. Und last not least müssen wir ja auch noch die beiden voll virtuellen Funktionen **"compare"** und **"ausgabe"** definieren, denn **"Eintrag"** soll ja kein abstrakter Datentyp mehr sein:

```
class Eintrag : public Listenelement
{ public:
    char name[30];      // Name und...
    char nummer[20]; // Telefonnummer

    // Konstruktor:
    Eintrag(const char *neuname, const char *neunummer);

    // Funktionen, die virtuelle Funktionen der Basisklasse
    // definieren:
    int compare(const Listenelement*) const;
    void ausgabe() const;
}
```

Auch hier wollen wir die fraglichen Funktionen gleich definieren. Zunächst einmal wäre da der Konstruktor, der ganz einfach seine beiden Argumente in das **"Eintrag"**-Objekt kopieren muß:

```
Eintrag::Eintrag(const char *neuname, const char *neunummer)
:name(neuname), nummer(neunummer) { }
```

Beachtenswerterweise müssen wir hier nicht auf **"strcpy"** wie im ersten Telefonprogramm zurückgreifen, denn bei dieser Schreibweise haben wir es mit einer echten Initialisierung und nicht mit einer Wertzuweisung zu tun, und folglich ist der Compiler gnädigerweise bereit, die Strings ganz einfach so zu initialisieren.

Die Funktion **"compare"** soll das Objekt, auf dem es aufgerufen wird, mit dem Element, auf das ihr Argument zeigt, vergleichen und in Abhängigkeit davon -1, 0 oder +1 zurückgeben. Das läßt

sich im Prinzip mit einem einfachen `"strcmp"` der Namensstrings bewerkstelligen, wobei wir nur darüber stolpern, daß eine Funktion eine virtuelle Funktion nur dann überdeckt, wenn die Argumente absolut identisch sind. Also muß die Funktion hier ein `"Eintrag"` (nämlich `"*this"`) mit einem `"Listenelement"` (das Objekt, auf das das Argument zeigt) vergleichen. Hier müssen wir ganz einfach darauf vertrauen, daß alle Elemente der Liste zur Klasse `"Eintrag"` gehören, und uns mit einer simplen Typkonvertierung behelfen:

```
int Eintrag::compare(const Listenelement* that) const
{ Eintrag *that2 = (Eintrag*) that;
  return strcmp(this->name, that2->name);
}
```

Ich hoffe, daß Sie schon jetzt das nötige „Unrechtsbewußtsein" aufbringen, um diese Funktion nicht so recht zu mögen. Wir haben hier einen Zeiger auf ein „Listenelement" und vertrauen darauf, daß es sich in Wirklichkeit um einen „Eintrag" handelt, und zwar nur deshalb, weil wir nicht vorhaben, Objekte anderer Klassen in die Liste einzutragen. Aber andererseits ist es ja auch wenig sinnvoll, einen Telefonbucheintrag mit einem Regenwurm oder einem spanischen Großinquisitor zu vergleichen, und deshalb ist es durchaus sinnvoll, für die Funktion `"compare"` anzunehmen, daß alle Listenelemente gleicher Art sind.

Außerdem ist es eine verblüffende (und in dieser Form vielleicht nicht für alle Anwendungsgebiete verwendbare) Tatsachen daß wir die Funktion `"compare"` in unserem Datemmodell für zwei verschiedene Zwecke verwenden: Erstens zum Suchen eines Elemente (Test auf Gleichheit) und zweitens zum Sortieren beim Einfügen (wo es im Wesentlichen auf „kleiner" oder „größer-gleich" ankommt). Schließlich weiß jedes Kind, daß es in Los Angeles mindestens drei verschiedene „Sarah Connor" gibt...

Die folgende Funktion ist wenigstens halbwegs unkompliziert: Sie gibt ein Objekt der Klasse `"Eintrag"` aus:

```
void Eintrag::ausgabe() const
{ cout << name << ": " << nummer << "\n";
}
```

Jetzt fehlt uns nur noch die Spezialisierung der Klasse `"Liste"` zum „Telefonbuch". Da wir es bei den Eigenschaften der Klasse `"Liste"` hauptsächlich mit Low-Level-Features zu tun haben, besteht kein Bedarf, diese Basisklasse öffentlich zu machen. Statt dessen vererben wir hier mit Modus `"private"` und machen nur die Member-Funktionen `"ausgabe"` und `"loeschen"` durch Zugriffsdeklarationen `"public"`.

Auf die Member-Funktion `"Liste::insert"` sollte der Benutzer der Klasse nicht zugreifen können, weil er dann ja theoretisch einen Fadenwurm in ein Telefonbuch einfügen könnte, wie Sie sich vielleicht erinnern. Also bleibt `"Liste::insert"` in der privaten Basisklasse verborgen, und wir bieten dem Klassenbenutzer eine Member-Funktion `"Telefonbuch::insert"` an, die als Argumente zwei Strings erwartet und einen entsprechenden Telefonbucheintrag erzeugt sowie in die Liste einfügt. Auch die Funktion `"suchen"` definieren wir neu, um erstens ganz einfach nach

einem Namen suchen zu können (und nicht anhand eines Vergleichselements) und zweitens die Typgleichheit zu garantieren. Daß wir mit "insert" und "suchen" die entsprechenden Funktionen der Basisklasse namensgleich überdefinieren, hat übrigens keine besondere Bedeutung, denn die gleichnamigen Funktionen der Basisklasse sind ja sowieso "protected" und deshalb von außen nicht erreichbar.

Hier ist also die letzte Klassendefinition unseres Programms:

```
class Telefonbuch : private Liste
{ public:
     Eintrag *insert(const char *, const char *);
     Eintrag *suchen(const char *);
     Liste::ausgabe;
     Liste::loeschen;
};
```

Die beiden Member-Funktionen, die es jetzt noch zu definieren gilt, sind vergleichsweise dröge, da sie hauptsächlich auf Funktionen der Basisklasse "Liste" zurückgreifen - und genau das war ja auch unsere Absicht, denn wenn wir einmal den abstrakten Datentypen "Liste" implementiert haben, wollen wir schließlich ohne größeren Aufwand spezielle Listentypen ableiten können. Zuerst also die neue "insert"-Funktion, die nur noch mit "new" ein neues Element erzeugt, es in die Liste einfügt und einen Zeiger darauf (bzw. Null bei Fehler) zurückgibt:

```
Eintrag* Telefonbuch::insert(const char *neuname, const char
*neunummer)
{  // Neues Element erzeugen:
   Eintrag *neu = new Eintrag (neuname, neunummer);

   if (!neu)
     return 0; // Kein Speicher frei!

   // Element in Liste einhängen:
   Liste::insert(neu);
   // Fertig, Zeiger auf neues Element zurückgeben:
   return neu;
}
```

Die Suchfunktion ist noch viel einfacher: Hier erzeugt man aus dem Suchstring ein Vergleichsobjekt (als automatische Variable!) und sucht dieses Element mit der vorhandenen Funktion in der Liste. Nun muß das Ergebnis nur noch mit einem Cast umgewandelt werden, da "Liste::suchen" einen Zeiger auf ein „Listenelement" zurückgibt, hier aber sinnvollerweise ein Zeiger auf "Eintrag" geliefert werden soll, und schon sind wir fertig:

```
Eintrag *Telefonbuch::suchen(const char *name)
{  Eintrag dummy(name," ");
   return (Eintrag*)
   Liste::suchen(&dummy);
};
```

Ob Sie es glauben oder nicht - das war's! Jetzt folgt noch einmal das ganze Programm im Überblick, einschließlich eines kleinen Hauptprogramms, daß die Benutzung unserer Klassen und Funktionen demonstriert.

```
//  Objektorientiertes Telefonlisten-Programm //
//  mit generischen Listenklassen //


#include <stream.h>
#include <string.h>

// * * * abstrakter Datentyp "Listenelement" * * *

class Listenelement
{ private:
     Listenelement *next, *prev; // Vorgänger und Nachfolger

  virtual int compare(const Listenelement*) const = 0;
  // Vergleicht Objekt "*this" mit einem
  // anderen Listenelement
  // und gibt -1, 0 oder +1 zurück (wie bei "strcmp")

  virtual void ausgabe() const = 0;
  // Ein Listenelement ausgeben

  virtual ~Listenelement()  { }
  // Virtueller Destruktor - hier nur ein Dummy, kann aber
  // in abgeleiteten Klassen überdeckt werden

  protected:
  // Kleine Extras: abgeleitete Klassen dürfen
  // "next" und "prev" auslesen:
  Listenelement *getnext() const
  { return next;
  }
  Listenelement *getprev() const
  { return prev;
  }

  friend class Liste;  // Diese später deklarierte Klasse
       // braucht Zugriff auf private Member
};

  // * * * generischer Datentyp "Liste" * * *

class Liste
{ private:
     Listenelement *anfang, *ende;
     // Erstes und letztes Element der Liste
  protected:
     void insert(Listenelement *neu);
     // Sucht eine geeignete Position für das
```

```
    // Element "*neu" und
    // hängt es in die Liste ein

  public:
    Liste(); // Konstruktor zur Initialisierung
    ~Liste();   // Destruktor (löscht alle
                // Listenelemente)
    void ausgabe(); // gibt die ganze Liste aus

    Listenelement *suchen(Listenelement *);
    // Sucht ein Element, das dem Argument-Element
  // "entspricht" (was immer das in einer konkreten Liste
  //  heißen mag - jedenfalls ein Element, bei dem
  // "compare" Null liefert) oder Null, wenn kein solches
  //  Element existiert.

  void loeschen(Listenelement*);
  // Hängt ein Element aus der Liste aus und löscht es
  // mit "delete"
  };

  Liste::Liste()
  { anfang = 0;
    ende = 0;
}

Liste::~Liste()
{ Listenelement *p = anfang;

  while(p)
  { Listenelement *hilf = p;          p=p->next;
  delete hilf;
  }
}

void Liste::ausgabe()
{ for(Listenelement *l = anfang; l; l = l->next)
    l->ausgabe();
}

Listenelement *Liste::suchen(Listenelement *e1)
{ for(Listenelement *e2 = anfang;
  e2 && e2->compare(e1);
  e2 = e2->next);
  return e2;
}

void Liste::insert(Listenelement *neu)
{ // Position suchen:
  Listenelement *pos = anfang;
  while (pos && neu->compare(pos) > 0)
    pos = pos->next;
```

```
  // Element an Position "pos" in Liste einfügen
  // (prinzipiell genau wie in der ersten Programmversion)

    if (pos == 0) // Sonderfall: Anfügen an das Ende der Liste
    { Listenelement *alt = ende;   // Altes Listenende

      if (alt != 0) // Allgemeiner Unter-Fall: Wirklich anhängen
      { alt->next = neu;
        neu->prev = alt;
        neu->next = 0;
        ende = neu;
      }
      else // Spezieller Spezialfall: Liste ist noch leer
      { neu->next = 0;
        neu->prev = 0;
        anfang = neu;
        ende = neu;
      }
    }
    else
      if (pos->prev == 0)
      // Sonderfall: Neues Element an Listenanfang hängen
      { pos->prev = neu;
        neu->next = pos;
        neu->prev = 0;
        anfang = neu;
      }
      else // Allgemeiner Fall: Element in Liste einhängen
      { Listenelement *vor = pos->prev;
        // "neu" zwischen "vor" und "pos" einhängen:
        vor->next = neu;
        neu->prev = vor;
        neu->next = pos;
        pos->prev = neu;
      }
    }

void Liste::loeschen(Listenelement *E)
// Löscht das Element, auf das "E" zeigt, aus der Liste
{   // Vorgänger und Nachfolger des zu löschenden Elements
  Listenelement *vor = E->prev, *nach = E->next;

  if (vor)
    // Element hat Vorgänger: Dann dessen Nachfolger umsetzen
    vor->next = nach;
  else
    // Kein Vorgänger: Dann Listenanfang aktualisieren
    anfang = nach;

  if (nach)
  // Element hat Nachfolger: Dann dessen Vorgänger verändern
```

```
    nach->prev = vor;
  else
  // kein Nachfolger, also Listenende anpassen
    ende = vor;

  delete(E);
}

// * * * spezieller Datentyp "Eintrag" * * *

class Eintrag : public Listenelement
{ public:
    char name[30];// Name und...
    char nummer[20];    // Telefonnummer

        // Konstruktor:
    Eintrag(const char *neuname, const char *neunummer);

  // Funktionen, die virtuelle Funktionen der Basisklasse
  // definieren:
    int compare(const Listenelement*) const;
    void ausgabe() const;
}

Eintrag::Eintrag(const char *neuname, const char *neunummer) :
name(neuname),      nummer(neunummer) { }

int Eintrag::compare(const Listenelement* that) const
{ Eintrag *that2 = (Eintrag*) that;
  return strcmp(this->name, that2->name);
}

void Eintrag::ausgabe() const
{ cout « name « ": " « nummer « "\n";
}

// * * * spezieller Datentyp "Telefonbuch" * * *

class Telefonbuch : private Liste
{ public:// Neues Element erzeugen und einfügen:
    Eintrag *insert(const char *, const char *);

  // Element anhand des Namens suchen:
    Eintrag *suchen(const char *);

  // Zwei Funktionen werden öffentlich vererbt:
    Liste::ausgabe;
    Liste::loeschen;
};

Eintrag* Telefonbuch::insert(const char *neuname, const char
*neunummer)
```

```
{ // Neues Element erzeugen:
  Eintrag *neu = new Eintrag (neuname, neunummer);

  if (!neu)
    return 0; // Kein Speicher frei!

  // Element in Liste einhängen:
  Liste::insert(neu);

  // Fertig, Zeiger auf neues Element zurückgeben:
  return neu;
}

Eintrag *Telefonbuch::suchen(const char *name)
{ Eintrag dummy(name,"");
  return (Eintrag*)
  Liste::suchen(&dummy);
};

// * * *  Hauptprogram  * * *

void main()
{ // Eine Liste wird deklariert und initialisiert:

  Telefonbuch t;

  // Ein paar Beispieldaten werden eingefügt:

  t.insert("Harley-Davidson", "001-414/342-4680");
  t.insert("Boris Becker", "0815/4711");
  t.insert("James Bond", "007/26731");
  t.insert("Maxon", "06196/481811");
  t.insert("Zaphod Beeblebrox", "00042/08154242");
  t.insert("Luxemburg", "00352");
  t.insert("Queensr\0377che", "795069");
  t.insert("Fishbone", "4676152");

   // Ein Datensatz wird gesucht und ggf. gelöscht:

  Eintrag *e = t.suchen("Maxon");
  if (e)
    t.loeschen(e);
  else
    cout « "Name nicht gefunden!\n";

  // Restliche Liste ausgeben:
  t.ausgabe();

  // Löschen der Liste geschieht automatisch durch Destruktor-Aufruf
}
```

3.5 Noch mehr Features

3.5.1 Zeiger auf Member

Zeiger auf Member stellen eigentlich kein objektorientiertes Feature dar, obwohl sie in das objektorientierte Konzept eingebunden sind. Man könnte sie sich durchaus auch als Ergänzung zu C vorstellen.

Die Idee ist, daß man bisweilen nicht über einen Member eines ganz bestimmten Klassen- bzw. Strukturobjekts, sondern nur über einen Member an sich „sprechen" will. Rein implementatorisch ist ein Member-Pointer kein wirklicher Zeiger, sondern der Offset eines bestimmten Members innerhalb einer Struktur, und konkret deklariert man solche Datentypen wie folgt:

```
struct S
{ int i, j;
  char *s;
  int k;
};

S s1, s2, s3;

int S::* pims;
```

Diese überaus komplizierte Syntax deklariert `"pims"` als Zeiger auf einen `"int"`-Member von `"S"`. Prinzipiell werden Zeiger-auf-Member-Typen wie gewöhnliche Zeiger deklariert, nur daß eben der `"*"` mit einem Klassennamen qualifiziert wird. So weit, so gut, fehlt nur noch der praktische Nutzen derartiger Zeiger. Wir können die Variable `"pims"` jetzt mit einem Member von `"S"` initialisieren, aber nicht mit einem Member eines bestimmten Objekts, sondern nur mit einem Member-Namen, und der wird naturgemäß qualifiziert, und natürlich muß man, weil es sich ja angeblich um einen Zeiger handelt, dazu den unären Operator `"&"` verwenden:

```
void main()
{ pims = &S::i; // Fortsetzung folgt...
```

OK, jetzt enthält `"pims"` (falls Sie sich immer noch über diesen Namen wundern: das soll „Pointer to Int-Member of S" heißen) eine interne Repräsentierung des Offsets von `"i"` in `"S"`. Jetzt können wir über `"pims"` auf Member von beliebigen `"S"`-Objekten zugreifen, und weil `"pims"` so initialisiert ist, erwischen wir jeweils den Member `"i"`:

```
  s1.*pims = s2.*pims;
}
```

Diese Anweisung kopiert also einen `"int"`-Member von `"s2"` in einen Member von `"s1"`, und da `"pims"` so initialisiert ist, handelt es sich jeweils um den Member `"i"`. Dabei ist `".*"` nicht eine Zusammenstellung von `"."` und `"*"`, sondern ein eigenständiger binärer Operator. Analog dazu gibt es auch noch `"->*"`, wobei

```
x->*y
```

und

```
(*x).*y
```

einander entsprechen, z. B.:

```
S *sptr = &s3;

sptr->*pims = foo;
```

Wozu das alles gut sein soll? Nun, mit Pointern auf Member kann man z. B. Member so ähnlich wie Vektor-Elemente behandeln:

```
struct S
{ int foo, bar, sucker;
};

int S::* nr(int i) // Gibt Zeiger auf den "i"-ten Member
                   // von "S" zurück
{ switch (i)
  { case 0:
      return &S::foo;
    case 1:
      return &S::bar;
    case 2:
      return &S::sucker;
    default: // Panik
  }
}

void main()
{ S s1;

  // Initialisiere "s1" wie einen Vektor:
  for (int i=0; i<3; ++i)
    s1.*nr(i) = 42 + i;

}
```

Es gibt wie bei normalen Zeigern einen ausgezeichneten Wert, nämlich `"0"`, den ein Member-Zeiger niemals auf „normalem" Wege annehmen kann, d. h. es ist garantiert, daß `"&x::y"` immer einen von Null verschiedenen Wert liefert. Deshalb kann man die (eigentlich numerische) Konstante `"0"` benutzen, um anzuzeigen, daß ein Member-Pointer momentan auf „nichts" zeigt, wie im folgenden (wenig sinvollen) Beispiel:

```
struct S    { char *cp1, cp2; };

char * S::* stringmemberA = &S::cp1, * S::* stringmemberB = 0;

void f(S *sp)
{ if(stringmemberA != 0 && stringmemberB == stringmemberA)
```

```
    sp->*stringmemberA = "Oh no, Babe!";
}
```

Rein praktisch wird das in MaxonC++ so gelöst, daß ein Pointer auf Member nicht direkt durch den Offset dieses Members in der Klasse bzw. Struktur dargestellt wird, sondern um 1 größer ist, d. h. ein Zeiger auf den allerersten Member einer nicht abgeleiteten Struktur hat den Wert 1. Übrigens sehen Sie oben auch, daß man Member-Zeiger mit den üblichen Operatoren vergleichen kann. Additionsoperatoren wie bei „normalen" Zeigern gibt es hier aber nicht - logisch, denn eine Struktur ist kein Vektor, und deshalb kann man nicht davon ausgehen, daß vor oder hinter einem Member ein weiterer Member gleichen Typs liegt.

Ansonsten wäre noch zu erwähnen, daß die objektorientierten Konzepte hier natürlich ebenfalls unterstützt werden. Im folgenden Beispiel sehen Sie, daß man einen Pointer auf einen Member einer Basisklasse direkt in einen Zeiger auf einen Member einer abgeleiteten Klasse umwandeln kann, während man für die umgekehrte Konvertierung einen Cast braucht:

```
struct Base
{ double d;
};

class Klasse: public Base
{ public:
    double e;
};

Base b;   Klasse c;

double Klasse::* kmp = &Base::d;

double Base::* bmp = (double Base::*) &Klasse::e;
```

Die Typregeln sind also genau umgekehrt wie bei Zeigern, wo man einen Zeiger auf eine abgeleitete Klasse ohne weiteres als Zeiger auf die Basisklasse verwenden kann und im umgekehrten Fall casten muß. Diese Regelung ist so aber durchaus sinnvoll, denn jede „Klasse" enthält eine „Basis", weshalb man einen Zeiger auf einen „Basis"-Member gewiß ohne Gefahr in einem „Klasse"-Objekt anwenden kann, während die Benutzung eines Zeigers auf einen „Klasse"-Member in einem „Basis"-Objekt voraussetzt, daß wir über die wahre Natur der Objekte, auf die wir diesen Zeiger später anwenden wollen, Bescheid wissen - es muß nämlich nicht nur eine „Basis", sondern sogar eine „Klasse" sein. Folgerichtig wird für diese haarige Typwandlung ein Cast verlangt.

Bei virtueller Vererbung funktionieren derartige Umwandlungen überhaupt nicht:

```
class B
{ public:
    int i;
};

class C: public virtual B
{ public:
```

```
    int j;
};

int B::* bmp = &B::i;

int C::* cmp = bmp;   // ERROR
```

Das ist, wenn man es implementatorisch sieht, auch völlig klar: Die Basisklasse `"B"` hat in `"C"` keinen festen Offset, statt dessen gibt es dort einen Zeiger auf die virtuelle Basisklasse, die weiß-der-Geier-wo liegen kann. Bei gewöhnlichen Adressumwandlungen, bei denen man es mit konkreten Objekten zu tun hat, ist das kein Problem, denn da kann man sich ja an diesen Zeigern entlanghangeln. Zeiger auf Member beziehen sich aber nicht auf ein konkretes Objekt, sondern auf eine Klasse an sich und können ohne jegliche existierende Klasseninstanz benutzt werden, also fällt das Verfolgen von Zeigern hier naturgemäß wohl aus.

Wenn Sie mich fragen, Zeiger auf Member sind die Lösung, nur hat noch niemand das Problem dazu gefunden. Es ist ein nettes kleines Feature, das man in der Praxis aber wohl von allen C++-Spielzeugen am seltensten brauchen wird.

3.5.2 Konvertierungsfunktionen

Ein Konstruktor mit einem einzelnen Argument kann bekanntlich einen beliebigen Ausdruck in ein Klassenobjekt verwandeln, etwa so:

```
class Eugen
{ int i;
  public:
    Eugen(int j)
    { i = j;
    }
};

void main()
{ int x = 1992;
  Eugen E1 = x;     // (1)
  E1 = x+2;           // (2)
}
```

Bei der Initialisierung (1) wird der Konstruktor direkt auf das Zielobjekt angewandt (d. h. `"this"` zeigt auf `"E1"`), während (2) eine gewöhnliche Wertzuweisung ist, weshalb der Compiler hier ein temporäres Objekt einführt, das er mit dem Wert `"x+2"` initialisiert, bevor er es in das eigentliche Zielobjekt `"E1"` kopiert (wie Sie sich vielleicht erinnern, unterscheidet C++ scharf zwischen Initialisierungen und Wertzuweisungen).

Weil das so toll ist (immerhin kann man ein Programm durch solche implizit aufgerufenen, anwender-definierten Typwandlungsfunktionen beliebig unübersichtlich gestalten), stellt sich natürlich die Frage, ob das vielleicht nicht auch umgekehrt geht, also eine Umwandlungsfunktion von einer Klasse nach einem beliebigen anderen Datentypen.

Im Prinzip will man also einen Konstruktor für "int" einführen, der als Argument ein Objekt der Klasse "Eugen" erwartet. Gegen diese Idee sprechen aber mindestens zwei Punkte: Erstens fällt einem dafür auf Anhieb keine anschauliche Syntax ein (Konstruktoren werden immer in der Klassendefinition, und wo zum Geier "definiert" man den Standardtypen "int") und zweitens und vor allen Dingen könnte dann jemand auf die Idee kommen, z. B. die Umwandlung zwischen "int" und "double" auf eine eigene originelle Methode zu definieren, was der Konsistenz des Sources natürlich nicht gerade zuträglich ist. Also geht man einen etwas anderen Weg: In einer Klasse kann man Konvertierungsfunktionen deklarieren, die ein Objekt dieser Klasse in einen beliebigen anderen Datentypen umwandeln und vom Compiler auch implizit, d. h. automatisch und ohne besondere Aufforderung, aufgerufen werden.

Der „Name" einer Konvertierungsfunktion besteht aus dem Wortsymbol "operator" und der Beschreibung des Typs, in den umgewandelt werden soll. Obwohl das in gewissem Sinne der Ergebnistyp der Funktion ist, darf man keinen Ergebnistypen explizit deklarieren, und natürlich hat eine Konvertierungsfunktion keine Argumente außer dem "this"-Zeiger auf ein Klassenobjekt. Ein Beispiel:

```
class HinUndHer
{  char test[20];
   public:

      HinUndHer() // Default-Konstruktor
      {  test[0] = 0;
      }

      HinUndHer(const char *init)  // Noch ein Konstruktor
      : test(init)
      {  }

      operator char*() // Konvertierung nach char*
      {  return test;
      }

      operator int();     // Noch eine Konvertierung

};

#include <string.h>
#include <stream.h>

HinUndHer::operator int()
{  return strlen(test);
}

void main()
{  int i;
   char *cp;

   HinUndHer h1 = "Beispiel", h2 = "Test";
```

```
    cp = h1;  // cp zeigt auf "Inhalt" von h1

    i = h2;// Länge des "Inhalts" von h2

    cout « cp « i;

}
```

Dieses Programm gibt `"Beispiel4"` aus. Ohne daß man einen Cast oder einen anderen Hinweis zur Typkonvertierung angeben müßte, wandelt der Compiler `"HinUndHer"`-Objekte nach `"int"` oder `"char*"` um, indem er klammheimlich die entsprechenden Konvertierungsfunktionen aufruft.

Natürlich werden solche Funktionen (wie alle anderen Member-Funktionen auch) vererbt und können virtuell sein.

Es kann (theoretisch) vorkommen, daß der Compiler sich nicht zwischen einer Konvertierungsfunktion und einem Konstruktor entscheiden kann:

```
class A;

class B
{ public:
     B(A&);
     B();
};

class A
{ public:
     operator B();
};

void main()
{ A a;
  B b;

  b = a; // Konstruktor oder Konvertierung???
}
```

In diesem (relativ krankhaften) Fall meldet der Compiler einen entsprechenden Fehler.

Außerdem ruft er beim Versuch einer Typwandlung höchstens eine Konvertierungsfunktion auf und nicht zwei nacheinander:

```
class X
{ public:
     operator long int();
};

class Y
{ public:
```

```
    operator X();
};

void main()
{ Y y1;
  long int l1 = y1,  // ERROR
  l2 = X(y1),          // So geht's
  l3 = y1.operator X().operator long int();
}
```

In solchen Fällen muß man sich eben behelfen, indem man (wie oben auf zwei verschiedene Arten geschehen) dem Compiler ausdrücklich klar macht, wie er die Typen ineinander umzuwandeln hat.

3.5.3 Temporäre Objekte

Mehrfach war schon die Rede von „temporären Objekten", die der Compiler ohne Zutun des Programmierers erzeugt, wenn es gerade nötig ist. Eigentlich gibt es darüber fast nichts zu sagen, denn erstens ist es in hohem Maße implementationsabhängig, ob und wann ein bestimmter Compiler ein solches Objekt erzeugt und wieder löscht, und zweitens ist die ganze Angelegenheit so konzipiert, daß alles so funktioniert, wie der Programmierer es erwartet. Nur manchmal wird es tückisch, und deshalb soll hier doch etwas näher auf das Thema eingegangen werden.

Ein temporäres Objekt ist eine Variable eines Struktur- bzw. Klassentyps, die der Compiler einrichtet, um kurzfristig ein Objekt zwischenzuspeichern. Naheliegenderweise werden sie von MaxonC++ im automatischen Speicher, also auf dem Stack, eingerichtet, aber solche Details sind implementationsabhängig. Der Programmierer kann sich aber darauf verlassen, daß temporäre Objekte genau wie alle anderen behandelt werden, insbesondere daß sie mit den jeweils richtigen Konstruktoren initialisiert werden und daß nachher (falls vorhanden) ein Destruktor aufgerufen wird. Wann solche Destruktoraufrufe erfolgen oder der Speicherbereich, in dem das temporäre Objekt lag, wieder für andere Zwecke benutzt wird, ist aber implementationsabhängig und für den Programmierer wenig interessant. Theoretisch könnte ein Compiler temporäre Objekte jeweils mit `"new"` erzeugen und erst am Programmende kollektiv destruieren und ihren Speicher freigeben - ein Verfahren, das zwar alles andere als effektiv (im Laufe der Zeit können sich dabei zahllose „Leichen" ansammeln, die nichts als Speicher verbrauchen), aber durchaus erlaubt und überdies auf der sicheren Seite liegt. Keine Angst, MaxonC++ wählt natürlich ein wesentlich effektiveres Verfahren.

Vielleicht sollte ich Ihnen endlich verraten, wann temporäre Objekte typischerweise eingerichtet werden. Nun, im Prinzip müssen Sie immer damit rechnen. Bei einer so harmlosen Initialisierung wie

```
class C
{ // usw.public:
  C();
  C(const C&);  // Copy-Konstruktor
   ~C();
};
```

```
void main()
{ C c1;
  C c2 = c1;
}
```

kann es theoretisch passieren, daß der Compiler bei der Initialisierung von `"c2"` durch `"c1"`, wo eigentlich bloß der Copy-Konstruktor anzuwenden ist, 42 zusätzliche temporäre Objekte einrichtet, dann das erste mit `"c1"` initialisiert, das zweite mit dem ersten, das dritte mit dem zweiten, ... und schließlich `"c2"` mit dem 42-ten temporären Objekt initialisiert (natürlich stets unter korrekter Anwendung des Kopier-Konstruktors) und anschließend auf jedes der 42 Phantomobjekte den Destruktor anwendet. Das Beispiel ist natürlich etwas übertrieben, aber daß es „dumme" Compiler gibt, die in einer solchen Situation ein temporäres Objekt einführen, sollte man stets als Möglichkeit bedenken. Das klingt jetzt aber viel schlimmer als es ist, denn als Programmierer wird man Konstruktoren und Destruktoren rein intuitiv ohnehin so definieren, daß eine zusätzliche Initialisierung keine Probleme bereitet.

Jetzt ein Beispiel, wo jeder Compiler ein temporäres Objekt einführen muß:

```
struct S
{ int i;
};

S f();

void main()
{ int j = f().i;
}
```

Die Funktion `"f"` gibt, wie man so schön sagt, ein Objekt des Typs `"S"` zurück. In Wirklichkeit muß man ihr natürlich beim Aufruf sagen, wo sie das Ergebnis abzulegen hat, denn im Gegensatz zu skalaren Werten kann man eine Struktir nicht so ohne weiteres in einem Prozessorregister ablegen (OK OK, hier würde es natürlich zufällig klappen, aber lassen wir das). Also erzeugt der Compiler ein temporäres Objekt, in das `"f"` ihr Ergebnis kopieren kann, und ermöglicht dann den Zugriff auf den Member `"i"` dieses Objekts. Anschließend hat der Mohr seine Schuldigkeit getan und kann gehen, sprich: das Objekt kann irgendwann später gelöscht werden.

Damit wären wir auch schon (wieder) bei einem haarigen Thema, einer potenten Quelle schier unentdeckbarer Bugs und tiefster Frustration: Bisweilen existieren temporäre Objekte weniger lange, als einem lieb ist. Modifizieren wir das obige Beispiel doch ein wenig:

```
struct S
{ char str[20];
};

S f()
{ S erg = { "Saratoga" };
  return erg;
}
```

```
#include <stream.h>

void main()
{ char *cp = f().str;

  cout « cp;
}
```

Die Funktion `"f"` hat ein lokales Objekt namens `"erg"`, das sauber initialisiert und beim `"return"` in das Zielobjekt kopiert wird. In diesem Programm ist das Zielobjekt aber wieder ein temporäres, und nun wird der `"char"`-Zeiger `"cp"` auf einen Member dieses temporären Objekts gesetzt. Die Folgen können fatal sein, denn wenn wir `"cp"` später benutzen, z. B. mit `"cout"` ausgeben, existieren das temporäre Objekt und mit ihm seine Member höchstwahrscheinlich nicht mehr! So etwas kann zu netten Überraschungen führen und man kann so etwas eigentlich nur vermeiden, wenn man mit Funktionen, die ein Objekt zurückgeben, höllisch vorsichtig ist.

Herzlichen Glückwunsch - Sie haben das Kapitel über die objektorientierten Features von C++ überstanden! Es folgen noch ein kurzes Kapitel über Möglichkeiten und Grenzen des Überladens und ein noch kürzers Kapitel zum Thema Preprozessor, und schon ist die C++-Sprachbeschreibung komplett - also keine Panik!

4. Überladen

4.1 Überladene Funktionen

4.1.1 Was man darf, und was nicht

Überladene Funktionen wurden in diesem Handbuch schon einige Male erwähnt, aber noch nicht wirklich formal (also mit allen Regeln, die dazu gehören) behandelt. Also soll in diesem Kapitel genau davon Rede sein.

Im Prinzip kann man jeden Funktionsnamen beliebig überladen, nur mit der Einschränkung, daß Funktionen mit identischen Parameterlisten keine unterschiedlichen Ergebnisse haben dürfen:

```
int f(int);

char *f(int);          // ERROR
```

Im Zusammenhang mit überladenen Funktionen sollte man zwischen den Begriffen „Funktion" und „Funktionsname" unterscheiden. Ein Funktionsname ist ein Bezeichner, der für mehrere Funktionen eines Scopes stehen kann. In Wirklichkeit gibt es also keine überladenen Funktionen, sondern überladene Namen, und das hat auch Konsequenzen:

```
class B
{ public:
   int funk(int);
};

class C: public B
{ public:
   char *funk(char*);
};

void main()
{ C c1;
   c1.funk(42);   // ERROR
}
```

Die Klasse `"C"` erbt zwar aus `"B"` eine Funktion namens `"funk"`, aber da sie selbst einen identischen Funktionsnamen deklariert, wird `"B::funk"` von `"C::funk"` vollständig überdeckt, obwohl die Parameterlisten doch völlig unterschiedlich sind. Deshalb kennt der Compiler in `"C"` nicht mehr ohne weiteres `"B::funk"` und akzeptiert kein `"int"`-Argument mehr für `"funk"`.

Es gibt noch einige andere Problemfälle von Funktionen, bei denen der Compiler nicht herausfinden kann, welche Funktion gemeint ist:

```
void g();

void g(int i = 0);
```

```
void main()
{ g(); // ERROR
}
```

Beide Funktionen können formal ohne Argumente aufgerufen werden, so daß der Compiler beim Funktionsaufruf `"g()"` nicht weiß, welche von beiden er nehmen soll. Prinzipiell kann ein Compiler solche Konflikte schon bei der Funktionsdeklaration erkennen und schon dort einen Fehler melden. MaxonC++ ist aber nicht ganz so schlau und entdeckt erst bei einem derartigen Funktionsaufruf, daß da etwas nicht ganz richtig sein kann. Aber das gehört schon fast in den folgenden Abschnitt, der davon handelt, nach welchen Regeln der Compiler bei überladenen Funktionsnamen anhand der Argumente die richtige Funktion herausfindet:

4.1.2 Was der Compiler davon hält

4.1.2.1 Die Hierarchie der Typwandlungen

Der einfachste Fall liegt vor, wenn eine Funktion mit genau einem Argument aufgerufen wird. Zunächst ist, auch wenn es trivial ist, zu erwähnen, daß der Compiler nur die Funktionen dieses Namens betrachtet, die auch mit einem einzelnen Argument aufgerufen werden können, d. h. solche Funktionen, die

- genau einen Parameter besitzen oder
- für alle Parameter (bzw. alle außer dem ersten) ein Default-Argument besitzen oder
- als Parameterliste nur die Ellipse `"(...)"` besitzen.

Nehmen wir also an, wir hätten einen geradezu üppig überladenen Funktionsnamen `"f"`:

```
void f();
int  f(int, int);
int  f(int);
void f(double);
void f(...);
short  f(char*, ...);
char f(char c1, char c2 = '?');
double f(float);
double f(struct S*);
```

Bei einem Funktionsaufruf wie `"f(x)"` für einen beliebigen einzelnen Ausdruck `"x"` bleiben davon folgende Funktionen im Rennen:

```
f(int);
f(double);
f(...);
f(char*, ...);
f(char c1, char c2 = '?');
f(float);
f(struct S*);
```

Die Ergebnistypen habe ich hier absichtlich weggelassen, denn sie spielen in der weiteren Analyse keine Rolle.

So weit, so cool. Bekanntlich gibt es in C++ eine Fülle von Typkonvertierungen, so daß ein Argument vom Compiler in einen passenden Parameter verschiedener Typen verwandelt werden kann. In einem Ausdruck wie `"f(0)"` könnte jede der oben aufgelisteten Funktionen gemeint sein. Um hier eine gewisse Klarheit zu schaffen, gibt es in C++ eine gewisse Hitliste der Typwandlungen:

1. Exakte Übereinstimmung der Datentypen
   Wenn es eine Funktion gibt, deren Parametertyp mit dem Argumenttyp identisch ist, macht diese natürlich das Rennen. Also würde `"f(17)"` zu einem Aufruf von `"f(int)"` führen. Dieser Fall ist gleichermaßen für den Compiler wie für den Programmierer der einfachste und eindeutigste. Wenn man sich gerade über die Typregeln nicht sicher ist, kann man natürlich immer mit einer expliziten Typwandlung, etwa `"f(int(x))"`, eine solche exakte Übereinstimmung erzwingen und ist so vor unliebsamen Überrschungen geschützt.

2. Triviale Typumwandlungen
   Die folgenden Umwandlungen gelten für jeden Datentypen „T" als trivial:

   ```
   T  -> T&
   T& -> T
   T[x] -> T*
   T  -> const T// x
   T  -> volatile T  // x
   T* -> const T*  // x
   T* -> volatile T* // x
   T(arg)  -> (*T)(arg)
   ```

   Die letzte Regel soll die Umwandlung eines Funktionsnamens in einen Zeiger auf eine Funktion symbolisieren.

   Eine Typwandlung, die ausschließlich diesen Regeln folgt, gilt als trivial, auch dann, wenn sie aus einer Folge solcher Konvertierungen besteht (z. B. `T& -> const T`).

   Es gibt hier noch eine Unterhierarchie: Die Umwandlungen, die oben mit einem `"x"` gekennzeichnet wurden, gelten als schlechter als die übrigen. Entsprechend sind auch Folgen trivialer Konvertierungen, die diese Konvertierungen enthalten, schlechter als solche, die ausschließlich aus den anderen bestehen.

3. Aufsteigende Konvertierungen
   Wie Sie sich vielleicht erinnern, rechnet C++ im wesentlichen mit 6 verschiedenen Datentypen, das sind

   ```
   int       unsigned int      long    unsigned long
   double    long              double
   ```

   Es steht einer Implementierung frei, auch in `"float"` Berechnungen auszuführen, wenn alle Operanden eines Operators `"float"` sind, und MaxonC++ tut dies auch. Ansonsten braucht sich der Compiler mit kleineren Typen gar nicht abzugeben und wandelt bei Berechnungen

Ausdrücke der „kleinen" Typen **"short"** und **"char"** (jeweils sowohl vorzeichenlos als auch vorzeichenbehaftet) in **"int"** um. Wenn eine Implementierung die oben erwähnten Berechnungen im **"float"**-GBereich nicht bietet, wandelt sie **"floats"** nach **"double"** um.

Damit man diesen Mechanismus auch bei überladenen Funktionen benutzen kann, gelten die Umwandlungen

```
char   unsigned char   signed char   -> int              short
unsigned short
```

sowie

```
float  -> double
```

als „aufsteigende Umwandlungen" (neudeutsch „Promotions") und stellen die dritte Stufe der Konvertierungshierarchie von C++ dar. Folgen von Typumwandlungen, die nur solche aufsteigenden Umwandlungen und eventuell auch die oben erwähnten trivialen Konvertierungen enthalten, sind besser als alle nachfolgend aufgelisteten Umwandlungen.

Was Sie als Programmierer davon haben? Nun, um eine numerische Funktion, z. B. den absoluten Betrag, auf allen numerischen Datentypen so zu definieren, daß es für jedes numerische Argument genau eine passende Funktion gibt, reicht es, wenn Sie sechs verschiedene Funktionen definieren, z. B. so:

```
int Abs(int);
unsigned Abs(unsigned);
long Abs(long);
unsigned long Abs(unsigned long);
double Abs(double);
long double Abs(long double);
```

In MaxonC++ sind natürlich **"int"** und **"long"** praktisch identisch und diese Unterschiedung ist nur für den Compiler von Bedeutung. Wenn Sie die speziellen Features von MaxonC++ nutzen wollen, kämen auch noch

```
long long Abs(long long);
unsigned long long Abs(unsigned long long);
```

dazu, aber das ist bekanntlich nicht Standard.
Natürlich haben Sie auch noch die Möglichkeit, die **"Abs"**-Funktion für weitere Parametertypen, z. B. **"short"**, zu definieren, ohne die Eindeutigkeit von Funktionsaufrufen zu beeinträchtigen: Bei **"Abs"** mit einem **"short"**-Argument würde eben jene zusätzliche Funktion aufgerufen, während **"unsigned short"**- oder **"char"**-Argumente weiter nach **"int"** expandiert würden.

Umgekehrt gibt es aber Probleme, wenn Sie einen der sechs Typen auslassen: Wenn z. B. **"Abs(unsigned)"** nicht deklariert wird, weiß der Compiler nicht, was er tun soll, wenn **"Abs"** mit einem **"unsigned"**-Argument aufgerufen wird, denn sowohl die Umwandlung

nach `"int"` als auch die nach `"long"` oder `"unsigned long"` fällt schon in die nachfolgende Kategorie (nebenbei bemerkt ist es natürlich ziemlich schwachsinnig, die Betragsfunktion auf vorzeichenlosen Datentypen zu definieren, aber es sollte ja auch bloß ein Beispiel sein).

4. Standard-Konvertierungen
   Diese Kategorie umfaßt alle vordefinierten Typumwandlungen, die oben noch nicht erwähnt wurden, nämlich

- alle sonstigen Umwandlungen zwischen numerischen Datentypen, beispielsweise von `"double"` nach `"signed char"` und was Ihnen noch so alles an möglichen Kombinationen einfällt,
- die Umwandlung der Konstanten „0" in einen Zeiger,
- die Konvertierung eines beliebigen Zeigertyps nach `"void*"`,
- Umwandlungen eines Zeigers auf eine Klasse in einen Zeiger auf eine direkte oder indirekte Basisklasse davon,
- die analoge Umwandlung zwischen Referenzen,
- Konvertierungen von Pointern auf Member in einen jeweils passenden Memberzeiger einer abgeleiteten Klasse.
- die Umwandlungen von einer abgeleiteten Klasse in eine ihrer Basisklassen.

   Bei den Zeigerumwandlungen gibt es aber noch eine Unter-Hierarchie: Wenn von einer Klasse `"A"` eine Klasse `"B"` und davon wiederum eine Klasse `"C"` abgeleitet wurde, ist die Umwandlung von `" C* "` nach `" B* "` besser als die von `" C* "` nach `" B* "`. Hier geht also die „Tiefe" einer solchen Umwandlung in ihre Güte mit ein. Für Referenzen auf Klassen sowie für Klassen selbst gilt analoges, und bei Zeigern auf Member ist es entsprechend umgekehrt, d. h. hier ist `"A::* -> B::*"` besser als `"A::* -> C::*"`.

   Damit wären alle Typkonvertierungen erwähnt, die der Compiler „von Natur aus" kennt. Typwandlungen, die in die Kategorie der Standard-Konveriterungen fallen, sind schlechter als die davor beschriebenen Konvertierungen, aber stets besser als vom Anwender (also dem Programmierer (also Ihnen)) deklarierte Konvertierungsfunktionen:

5. Anwenderdefinierte Konvertierungen
   Der Programmierer kann bekanntlich auch selbst Regeln zur Typwandlung deklarieren, und zwar in Form von Konstruktoren und Konvertierungsfunktionen. Der Compiler berücksichtigt bei Gelegenheit derartige Typwandlungen und führt bei Bedarf sowohl vorher als auch nachher Standard-Konvertierungen durch. Er versucht aber nicht, mehrere anwenderdefinierte Konvertierungen nacheinander auszuführen.

   Typwandlungen, die anwenderdefinierte Konvertierungen benutzen, sind schlechter als alle anderen, wobei es in der Hierarchie keinen Unterschied zwischen Konstruktoren und Konvertierungsfunktionen gibt.

Vielleicht fühlen Sie sich jetzt diskriminiert und zurückgesetzt, weil der Compiler „Ihre" Typkonvertierungen für „schlechter" als seine eigenen hält, obwohl Sie doch sicherlich intelligenter als so ein dummer Compiler sind. Das ist aber nicht wirklich so, denn in den meisten Fällen treten Sie hier gewissermaßen „außer Konkurrenz" an: Mit einem Konstruktor können Sie festlegen, wie ein beliebiger (z. B. numerischer) Wert in ein Klassenobjekt umgewandelt wird, und Konvertierungsfunktionen behandeln die umgekehrte Wandlung eines Klassenobjekts in irgendeinen anderen Ausdruck. Für solche Umwandlungen gibt es aber bekanntlich keine Standard-Konvertierungen.

Außerdem tröstet es Sie vielleicht, daß es bei Funktionsaufrufen auch noch eine „Konvertierung" gibt, die noch schlechter als ihre anwenderdefinierten ist:

6. Funktionsaufruf mit Ellipse
   Die Übergabe eines Arguments als „Ellipsen-Parameter" ist schlechter als alle anderen Parameterübergaben, auch wenn diese mit noch so schlechten Typkonvertierungen erfolgen. Das heißt in der Praxis, daß eine Funktion `"f(...)"` nur aufgerufen wird, wenn es wirklich keine andere Funktion namens `"f"` gibt, auf die das Argument irgendwie paßt.

Oft besteht eine Typkonvertierung aus mehreren Schritten, z. B. erst von `"int"` nach `"int&"` und dann nach `"const int&"`, was jeweils unter die Rubrik „triviale Umwandlung" fällt. Wenn der Compiler bewerten soll, welche von zwei Typkonvertierungen die Bessere ist, zählt ausschließlich der jeweils schlechteste Einzelschritt. Das hat bisweilen unangenehme Konsequenzen: Nehmen wir an, Sie versehen eine Klasse mit einer Konvertierungsfunktion nach `"const char*"`.

Da man Daten dieses Typs normalerweise mit `"cout"` ausgeben kann, liegt es nahe, das auch direkt mit einem Klassenobjekt zu tun:

```
#include <stream.h>

class C
{ char Inhalt[20];
  public:
    operator const char*();
};

void main()
{ C c1;
  cout « c1;  // ERROR: Ambiguous function call
}
```

Wieso das? Nun, mit `"cout"` kann man zwar `"const char*"`-Objekte, also Strings, ausgeben. Außerdem gibt es aber noch (neben vielen anderen, die uns hier nicht interessieren) die Möglichkeit, ein `"const void*"` auszugeben, nämlich die hexadezimale Adresse eines Objekts. Der Compiler hat nun zwei Möglichkeiten, `"c1"` in ein Argument für `"cout «"` umzuwandeln: Einerseits die kanonische, im obigen Beispiel vom Programmierer wohl gemeinte einfache Umwandlung

```
C ->const char*
```

und andererseits die Konvertierungsfolge

```
C ->const char* ->const void*
```

Überraschung, Überraschung: Für den Compiler sind diese Konvertierungen gleichwertig, denn es zählt ja jeweils nur der schlechteste Schritt, hier also die anwenderdefinierte Konvertierungsfunktion. Eine nachfolgende Standardkonvertierung, z. B. von `"const char*"` nach `"const void*"`, fällt dann nicht mehr ins Gewicht.

Es ist aber durchaus möglich, Klassen bequem und direkt auszugeben, indem man den Operator `"«"` überlädt. Dazu lesen Sie in 4.2.2.7 mehr.

4.1.2.2 Matching von Funktionsargumenten

Aus der obigen Hierarchie ergibt sich direkt, welche Funktion aufgerufen wird, wenn nur ein Argument übergeben wird: Der Compiler sucht sich aus allen Funktionen dieses Namens gerade die heraus, die nach dieser Hierarchie am besten paßt - oder sagen wir besser „passen", denn es kann durchaus auch mehrere Funktionen geben, die „gleich gut" sind:

```
void f(int);
void f(double);

void main()
{ int i;
  short s;
  unsigned u;

  f(i);   // OK, exakte Übereinstimmung mit "int"

  f(s);   // OK: aufsteigende Konvertierung nach "int", aber
          // nur Standard-Konvertierung nach "double"

  f(u);   // ERROR: ambiguous function call!
          // unsigned -> int und unsigned -> double sind
          // jeweils Standard-Konvertierungen, also gleich
          // schlecht!
}
```

So weit, so gut - aber was ist, wenn einer Funktion mehrere Argumente übergeben werden? Auch dann haben wir es mit einer ziemlich einfachen Regel zu tun, die man am besten mit einer Mengen-Terminologie (hat nichts mit Terminatoren zu tun) ausdrückt. Hier betrachtet der Compiler zunächst jeden Parameter einzeln und ermittelt dabei genau wie zuvor die Funktionen, bei denen dieses einzelne Argument am besten paßt. Das Ergebnis ist also eine Memge von Funktionen. Wenn er das für jedes Argument getan hat, bildet er die Schnittmenge über alle diese Mengen.

Oder, um es anders zu formulieren: Die Funktion, die insgesamt am besten paßt, ist gerade die, die bei jedem einzelnen Parameter zu den besten gehört. Es kann hier mehrere dieser Funktionen geben - oder auch gar keine.

Betrachten wir ein einfaches Beispiel mit zwei Funktionen, die jeweils drei Parameter besitzen:

```
int d(int, int, int);              // Funktion (a)
long d(long, double, double);  // Funktion (b)

void main()
{ int i1, i2, i3;
  char c1;
  unsigned u1, u2, u3;
  long l1, l2, l3;
  double d1, d2, d3;
  float f1, f2, f3;

  d (d1, c1, l1);  // (1) OK

  d (i1, d2, u3);  // (2) ERROR

  d (f1, f2, f3);  // (3) OK

  d (u1, l2, l3);  // (4) ERROR
}
```

Zwei der Funktionsaufrufe sind für den Compiler eindeutig und die beiden anderen eben nicht. Gehen wir doch einmal die vier Beispiele langsam und zum Mitdenken durch:

(1) Als erstes Argument wird ein `"double"` übergeben, und sowohl nach `"int"` als auch nach `"long"` haben wir es mit einer Standard-Konvertierung zu tun. Also ist die Funktionsmenge für das erste Argument `{a, b}`. Das zweite Argument vom Typ `"char"` kann mit einer aufsteigenden Konvertierung in ein `"int"` verwandelt werden, was besser als die Standard-Konvertierung nach `"double"` ist. Also ist die zweite Menge `{a}`, während wir beim letzten Argument wieder zwei Standard-Konvertierungen gleicher Güte vorliegen haben (Menge `{a, b}`). Die Schnittmenge von `{a, b}`, `{a}` und `{a, b}` ist bekanntlich `{a}`, und so haben wir hier einen eindeutigen Aufruf von Funktion `(a)` vorliegen.

(2) Das erste Argument paßt exakt zu Funktion `(a)`, während bei `(b)` eine Standard-Konvertierung nötig wäre. Also ist die beste Menge hier offensichtlich `{a}`. Beim folgenden `"double"`-Argument ist es genau umgekehrt: Es paßt exakt zur Funktion `(b)`, so daß die Funktionsmenge `{b}` herauskommt. Das dritte Argument paßt wiederum bei beiden Funktionen gleich schlecht (Standard-Konvertierung), weshalb wir hier `{a, b}` notieren. Die Schnittemenge von `{a} * {b} * {a, b}` ist aber leer - es gibt also keine Funktion, die bei allen Argumenten zu den besten gehört.

(3) Beim ersten Argument haben wir es mit zwei Standard-Konvertierungen zu tun (Menge `{a, b}`), während die beiden folgenden Argumente jeweils mit einer aufsteigenden Typwandlung in `"double"` verwandelt werden können (zweimal Menge `{b}`). Also ist `(b)` hier klarer Sieger.

(4) Bei allen drei Argumenten ist jeweils für beide Funktionen eine Standard-Konvertierung anzuwenden, weshalb für alle drei Mengen `{a, b}` herauskommt. Der Durchschnitt über dreimal diese Menge ist eben `{a, b}`, und so haben wir hier gleich zwei Funktionen die überall zu den am besten passenden gehören. Also ist dieser Funktionsaufruf mehrdeutig und deshalb ein ERROR.

Solche Analysen sind natürlich ziemlich krankhaft, aber das ist bei weitem nicht so schlimm, wie es vielleicht scheinen mag: In der Praxis kann man schließlich immer durch explizite Typkonvertierungen eindeutig und leicht ersichtlich vorgeben, was man meint.

4.2 Operatoren

4.2.1 Allgemeine Operatoren

4.2.1.1 Allgemeines über allgemeine Operatoren

Das mit den überladenen Funktionen ist ja ganz erbaulich, aber vielleicht fragen Sie sich trotzdem (zu Recht), was das Ganze soll. Man könnte in der Tat ohne überladene Funktionen auskommen, wenn, wenn es in C nicht schon ohnehin überladene Funktionen gäbe: Fast jeder Operator symbolisiert im Prinzip eine ganze Menge von ein- oder zweistelligen Funktionen. Beispielsweise steht das binäre `"+"` für nicht weniger als sechs oder sieben verschiedene Operationen, nämlich Addition von jeweils zwei `"int"`-, `"unsigned"`-, `"long"`-, `"unsigned long"`-, `"double"`- oder `"long double"`-Werten sowie in einigen Implementationen auch noch die `"float"`-Addition (und in MaxonC++ obendrein auch noch Additionen auf `"long long"` sowie `"unsigned long long"`). Außer diesen numerischen Additonen gibt es natürlich auch noch die Addition eines Pointers mit einer ganzen Zahl. Der Compiler braucht also sowieso Regeln, um aus den Argumenten auf die gewünschte Funktion zu schließen, und so sind überladene Funktionen gewissermaßen nur ein Feature, das mit den ohnehin vorhandenen überladenen Operatoren eng verwandt ist und deshalb leicht in den Sprachstandard integriert werden konnte.

Der einzige Unterschied ist bis jetzt, daß Operatoren vom Compiler fest vorgegeben sind, während Funktionen vom Anwender definiert werden. Aber genau das ändert sich mit diesem Abschnitt: Ab sofort ist ein `"+"` nicht meht bloß eine Addition, wie sie der Standard definiert - jetzt werden wir kreativ.

So gut wie jeder ein- oder zweistellige Operator kann vom Programmierer noch weiter überladen werden, als er es ohnehin schon ist. Sie kennen schon ein sehr nützliches Beispiel dafür: Die Operatoren `"«"` und `"»"`, mit denen man gewöhnlich Bits durch die Gegend schiebt, werden im Standard-Include „stream.h" auf den Klassen `"istream"` bzw. `"ostream"` überladen und wuchten plötzlich nicht mehr bloß binäre Daten durch den Speicher, sonderm bewirken höchst komplexe Ein- und Ausgabe-Operationen.

Aber bevor Sie in unangemessene Ekstase geraten, sei Ihre Euphorie ein wenig gedämpft, denn es gibt einige nicht unbedeutende Einschränkungen:

- Man kann keine völlig neuen Operatoren definieren, sondern ausschließlich den vorhandenen Operatoren zusätzliche Bedeutungen verleihen. So mancher würde z. B. einen Operator namens `"***"` oder `"pot"` für Potenzierungen einführen, aber das geht in C++ nicht (sagen wir besser: nicht einmal in C++).

- Weder Vorrang noch Bindung von Operatoren können verändert werden. Sie können also `"+"` und `"*"` soviel überladen, wie Sie wollen, aber Punktrechnung geht weiter vor Strichrechnung und die Bindung erfolgt dabei von links nach rechts, basta.

- Die Operatoren bleiben ein- oder zweistellig (d. h. sie besitzen ein oder zwei Parameter). Bei anwender-definierten Überladungen muß mindestens einer dieser Parameter ein Klassentyp oder eine Referenz auf eine Klasse sein. Das ist aber halb so schlimm, denn so vermeidet man, daß die von Ihnen definierten Operatoren mit den Standard-Operatoren kollidieren; sonst könnte z. B. ein Witzbold auf die Idee kommen, `"/"` mit einen `"double"` als linkem und `"unsigned short"` als rechtem Parameter eine ganz andere Bedeutung zu geben (zur Erinnerung: normalerweise wird ein solcher Ausdruck als Division in `"double"` ausgewertet).

Es war die Rede davon, daß man fast alle Operatoren überladen kann. Hier ist eine Liste der überladbaren Operatoren:

```
+ - * / % ^ & |
~ ! = < > += -= *=
/= %= ^= &= |= << >> >>= <<= == != <= >= && ||   ++ -- , ->* ->
() [ ] new delete
```

Dabei steht `"()"` für Funktionsaufrufe und `"[]"` für Indizierungen. Die Operatoren `"+"`, `"-"`, `"*"` und `"&"` können gleichermaßen als einstellige wie als zweistellige Funktione überladen werden, und `"++"` sowie `"--"` können als Postfix- oder Präfix-Operator unterschiedliche Bedeutungen erhalten.

Die Operatoren `"."` und `".*"` können nicht überladen werden, da sie auf Klassenausdrücken (zumindest auf der linken Seite) bereits eine vordefinierte Bedeutung haben umd man hier Kollisionen und Inkonsistenzen vermeiden will. Das Symbol `"::"`, das zur Compilezeit Scopes auflöst, ist nur im entferntesten Sinne als Operator aufzufassen und kann folglich ebenfalls nicht überladen werden, ebenso wenig wie der bedingte Ausdruck `"?:"`, der als dreistelliger Operator (Bedingung, `"if"`- und `"else"`-Teil) ein wenig aus der Reihe fällt. Es versteht sich fast von selbst, daß das `"#"` und `"##"`, die ausschließlich vom Preprozessor benutzt werden, gleichfalls nicht überladen werden können. Sonst ist in C++ aber alles überladbar, was nicht niet- und nagelfest ist, und wahrscheinlich sind Sie schon höllisch neugierig, wie man z. B. einen Funktionsaufruf überlädt, aber zuerst befassen wir uns hier mit eher gewöhnlichen Operatoren:

4.2.1.2 Wie man so etwas deklariert

Ein selbstdefinierter Operator ist zunächst einmal eine Funktion, die einen Namen hat. Die Funktion zu einem Operator "x" heißt "operator x" und wird auch als solche deklariert, z. B.

```
class X
{ public:
    int i;
};

int operator + (X x, int j)
{ return x.i + j + 42;
}
```

Und schon gibt es eine neue Addition, und zwar mit einem "X" als linkem und einem "int" als rechtem Operanden. Diese Operation ist jetzt aber nicht kommutativ, d. h. bei einer Addition "int + X" würde der Compiler diese neue Operation nicht aufrufen. Deshalb empfiehlt es sich in solchen Fällen normalerweise, die umgekehrte Operation auch noch zu deklarieren und definieren:

```
int operator + (int j, X x)
{ return x.i + j + 42;
}
```

Bei Operatorfunktionen muß die Anzahl der Parameter mit der Argumentanzahl der jeweiligen Standard-Operatoren übereinstimmen. Beispielsweise kann "operator -" wahlweise ein oder zwei Parameter haben (unäres bzw. binäres Minus!), während "/" immer genau zwei Parameter haben muß. Natürlich sind dann Default-Argumente und die Ellipse "..." verboten.

Davon einmal abgesehen, sind Operatoren ganz gewöhnliche Funktionen, die bloß etwas seltsam heißen ("operator xxx" statt eines Bezeichners) und auf eine noch seltsamere Art und Weise aufgerufen werden (siehe auch im nächsten Abschnitt). Sie können "friend" einer Klasse sein - oder auch ein Member:

```
class B
{ double d;
  public:
    B &operator += (double);
};

B& B::operator += (double x)
{ d += x;
  return *this;
}
```

In solchen Fällen dient der "this"-Zeiger stets als linker Operand, so daß der zweistellige Operator "+=" hier nur noch einen weiteren Parameter haben darf (und muß). Entsprechend dürfen einstellige Operatoren, wenn sie als Member deklariert werden, gar keine weiteren Parameter besitzen.

Wenn ein Operator als Member-Funktion einer Klasse deklariert wird, muß beim Funktionsaufruf der linke Operand naturgemäß ein L-Wert sein, auf den dann "this" zeigt (notfalls führt der Com-

piler aber auch ein temporäres Objekt ein). Es ist ziemlich egal, ob ein Operator als Member-Funktion oder auf Dateiebene deklariert wird, zumal auch die Scope-Regeln hier keinen direkten Einfluß haben (globale Operator-Funktionen überdecken gleichnamige Member nicht - das wäre ja auch fatal, denn es gibt jeden Operator sowieso schon als global vordefinierten Standard-Operator, so daß Member-Operator-Funktionen dann stets im Scope der Klasse versteckt und von außen überdeckt wären). Im Prinzip ist so etwas ganz einfach Geschmackssache, einmal davon abgesehen, daß einige Operatoren (z. B. `"="`) ausschließlich als Member-Funktion deklariert werden dürfen. Solche Sonderregelungen werden in Abschnitt 4.2.2 beschrieben.

4.2.1.3 Und wie man es benutzt

Da ein Operator eine (fast) normale Funktion ist, kann man ihn auch so aufrufen:

```
#include <stream.h>

class A
{ public:
     A operator ! ();
};

operator & (A& a1, A& a2);

void main()
{ A a, b;
  a.operator!();
  operator & (a, b);

  // in <stream.h> wird "«" auf Objekten der Klasse "ostream",
  // zu denen auch "cout" gehört, überladen, also könen wir
  // folgendermaßen Daten ausgeben:
  cout.operator « ("Hello, World!\n");

  // Allerdings ist jener Operator eine Member-Funktion, so daß
  // Folgendes nicht erlaubt ist:
  operator « (cout, "So geht's auch.\n"); // ERROR!
}
```

Das ist aber natürlich nicht der Sinn der Sache. Wesentlicn eleganter geht das nämlich so:

```
class A
{ public:
     A operator ! ();
};

A operator & (A& a1, A& a2);

void main()
{ A a, b, c;

  b = !a;
  c = a & b; }
```

Wenn Sie einen Operator benutzen, wendet der Compiler zum Auflösen der Überladung dieselben Regeln an, wie man sie von gewöhnlichen Funktionen kennt (vgl. 4.1.2), so daß es auch hier zu nicht eindeutigen Funktionsaufrufen kommen kann. Selbstdefinierte Operatoren sind gleichrangig mit vordefinierten.

4.2.2 Spezielle Operatoren

4.2.2.1 Der Zuweisungs-Operator „="

Das Kopieren von Klassenobjekten ist ein Vorgang ganz besonderer Art, weshalb der Copy-Konstruktor auch eine gewisse Sonderstellung einnimmt (z. B. wird er als einziger Konstruktor vom System defaultmäßig definiert). Allerdings wird dieser Konstruktor naturgemäß nur zur Initialisierung von Objekten eingesetzt, während er auf gewöhnliche Wertzuweisungen zwischen Objekten keinen Einfluß hat. Diese Lücke schließt man mit der Möglichkeit, den Operator `"="` zu überladen.

Der Operator `"="` darf ausschließlich als Member-Funktion definiert werden, z. B. wie im folgenden Beispiel, das ansatzweise (und auch nicht sehr geschickt) eine kleine String-Bibliothek implementiert:

```
#include <string.h>

class String
{ char *s;
  public:
    String &operator = (const String&);
    String &operator = (char*);
};

String &String::operator =(char* init)
{ delete s; // alten String-Inhalt löschen
  s = init;
  return *this;
}

String &String::operator = (const String &str)
{ delete s;                // Alten String löschen
  s = new char[strlen(str.s)+1]; // Neuen allozieren
  strcpy(s, str.s);        // Stringinhalt kopieren
  return *this;            // Referenz zurückgeben
}

void main()
{ String s1, s2;

  s1 = "Test";  // Entspricht s1.operator =("Test");
  s2 = s1;      // Entspricht s1.operator =(s1);
}
```

Normalerweise ist es das Sinnvollste, als Ergebnis eines Zuweisungsoperators das soeben initialisierte Objekt (also `"*this"`) bzw. eine Referenz darauf zurückzugeben, um so die Semantik einer normalen Wertzuweisung zu imitieren. Verlangt wird das aber keineswegs.

Der Zuweisungsoperator mit einer Referenz auf ein Klassenelement als Argument nimmt eine Sonderstellung unter den überladbaren Operatoren ein: Als einziger besitzt er eine defaultmäßige Definition. Zu jeder Klasse K gibt es einen Operator (oder zumindest so etwas ähnliches)

```
K &K::operator = (const K &),
```

der Klassenobjekte Member für Member kopiert. Der Programmierer hat dann die Möglichkeit, diesen Operator selbst zu deklarieren und dann zu definieren (wobei er auch das `"const"` weglassen darf). Dies ist ein bedeutender Unterschied zu allen anderen Operatoren, denn mit `"+"`, `"*"` oder `"+="` kann man ja lediglich Operationen, die es normalerweise nicht gibt, auf Klassen definieren.

Diese besondere Rolle hat einige Konsequenzen: Der Operator `"="` mit einer Referenz auf seine eigene Klasse ist der einzige Operator, der wirklich vererbt und in abgeleiteten Klassen bei Bedarf auch automatisch deklariert wird. Dies geschieht völlig analog zur Behandlung von Copy-Konstruktoren (3.3.4.2):

```
class Base
{ public:
     Base();
     Base(const Base&);
     Base &operator = (const Base&);
};

class Derived : Base
{ public:
     Derived();
};

void main()
{ Derived d1, d2(d1);
  d1 = d2;
}
```

In der Klasse `"Derived"` definiert der Compiler automatisch einen Kopier-Konstruktor

```
Derived::Derived(const Derived &)
```

sowie einen Zuweisungsoperator

```
Derived &Derived::operator = (const Derived &)
```

die jeweils zuerst den Kopier-Konstruktor bzw. Zuweisungsoperator der Basisklasse aufrufen und anschließend den Rest der Klasse ganz normal kopieren.

Von Compiler generierte Zuweisungsoperatoren haben den Zugriffsmodus `"public"` - auch das ist genau wie bei den Konstruktoren.

Auch wenn `"x += y"` bei Standard-Datentypen mit `"x = x + (y)"` identisch ist, hat die Definition eines eigenen `"="`-Operators auf einer Klasse keineswegs zur Folge, daß auch `"+="` oder ähnliche Operatoren automatisch aufgerufen werden können. Wenn man solche Operatoren auf einer Klasse definiert haben möchte, muß man das schon selbst tun. Diese müssen übrigens nicht Member einer Klasse sein:

```
class C    {      // ...    };

C &operator = (C&, const C&);          // ERROR

C &operator +=(C&, const C&);          // OK
```

Warum aber muß `"="` stets als Member deklariert werden? Nun, auf diese Weise stellt der Compiler sicher, daß die Gestalt des Zuweisungsoperators (für den es ja auch eine Default-Bedeutung gibt) fest steht, sobald die Klassendefinition abgeschlossen ist. Folgender Code wäre einfach zu grausam:

```
class Err   {      // ...    };

Err e1, e2;

void Falsch()
{ e1 = e2;  // Standard-Zuweisung
}

Err &operator = (const Err&, Err&); // ERROR

void TotalFalsch()
{ e1 = e2;     // Selbstdefinierte Zuweisung
}
```

Da der Compiler bekanntlich nicht hellsehen kann, würde er die erste Zuweisung für ein Standard-`"="` (also memberweises Kopieren) halten, während die zweite offensichtlich ein Aufruf der anwenderdefinierten Funktion ist. Entweder müßte der Compiler also einen unangemessenen Aufwand treiben, um solche Fälle als Fehler abzufangen bzw. bereits gelesene Zuweisungen nachträglich noch einmal zu betrachten (das gefällt dem Compilerbauer gar nicht), oder die Zuweisungsoperation wäre inkonsistent (das ist nichts für Ästheten). Also ist der Zwang, `"="` als Member zu deklarieren, die sinnvollste Lösung. Dagegen stellt sich dieses Problem bei `"+="` oder `"*="` nicht, denn diese Operatoren haben bekanntlich keine vordefinierte Bedeutung auf Klassen.

4.2.2.2 „new" und „delete"
Selbstgeschnitzte Speicherverwaltung

Bei `"new"` und `"delete"` werden gewöhnlich fest vordefinierte Funktionen aus den Bibliotheken des Compilersystems aufgerufen. Oft ist diese Speicherverwaltung aber nicht effektiv genug: Möglicherweise möchte man Objekte einer bestimmten Klasse zunächst immer nur allozieren und dann am Programmende auf einen Schlag vollständig löschen. Wenn man so etwas sowohl auf Laufzeit als

auch auf Speicherbedarf optimieren möchte, stellt man schnell fest, daß man mit der vollständigen Heap-Verwaltung eigentlich mit Kanonen auf Spatzen schießt. Schließlich müssen das Laufzeitsystem und das Betriebssystem des Rechners gleichermaßen umfangreiche Listenstrukturen verwalten, um sich zu merken, welchen Speicher das Programm reserviert hat und wo noch etwas frei ist. Das kostet einen nicht unerheblichen Overhead an Speicherplatz und Laufzeit.

Eine beliebte Lösung für das oben angedeutete Problem ist, sich einmal einen gewissen Speicherblock (sagen wir: 10 KByte) mit den Standard-Funktionen zu allozieren und in diesem Speicherbereich dann alle dynamisch erzeugten Objekte nacheinander abzulegen, bis er voll ist - und dann richtet man eben noch einen solchen Block ein. Da zwischendurch kein **"delete"** benutzt wird, braucht man keine wirkliche Heap-Verwaltung und kann deshalb sehr schnell Datenobjekte allozieren und braucht auch praktisch keinen Speicher-Overhead. Der MaxonC++ Compiler verwaltet fast alle bei einem Übersetzungsvorgang anfallenden Daten auf diese Weise, was dann auch nicht unerheblich zu der beeindruckenden Übersetzungsgeschwindigkeit beiträgt.

Auch sonst lassen sich sicher noch andere Gelegenheiten denken, bei denen die allgemeine Standard-Speicherverwaltung den aktuellen Erfordernissen nicht genügt. Lange Rede, schwacher Sinn: C++ unterstützt anwenderdefinierte Speicherverwaltungen durch Überladen der Operatoren **"new"** und **"delete"**.

Die Operatoren **"new"** und **"delete"** können als Member von Klassen deklariert werden, wodurch dann Objekte dieser Klasse (und aller abgeleiteten Klassen) mit diesen Operatoren alloziert und freigegeben werden. Dabei gibt es aber das kleine Problem, daß ein Programm bei einem **"new"**-Aufruf erst Speicher allozieren muß und erst dann den Konstruktor aufrufen kann, und umgekehrt den Speicher erst freigeben kann, nachdem der Destruktor ausgeführt wurde. Also operieren Speicherverwaltungsfunktionen nicht auf einem Objekt und sind deshalb grundsätzlich statische Member (d. h. ohne **"this"**-Zeiger), auch wenn man das nicht explizit so deklariert. Wie alle statischen Funktionen können **"new"** und **"delete"** folglich auch nicht virtuell sein.

Überladen von „new"

Im Include-File „*<stddef.h>*" steht ein **"typedef"**, das einen numerischen Datentypen unter dem Namen **"size_t"** definiert. Bei MaxonC++ handelt es sich dabei um den Typen **"unsigned int"**. Ein **"new"**-Operator muß natürlich wissen, wieviel Speicher er reservieren soll, und so muß der erste Parameter immer jenen Typen **"size_t"** haben:

```
#include <stddef.h>

class Beispiel
{ public:
     void *operator new(size_t Groesse);
     void *operator new(size_t Groesse, int Nochwas);
};
```

Der Ergebnistyp eines **"new"**-Operators muß stets **"void*"** sein, und das hat auch seine Richtigkeit, denn auch wenn der Operator benutzt wird, um Speicher für ein Objekt der Klasse **"Beispiel"** zu reservieren, und einen Zeiger auf den allozierten Speicher (bzw. 0 bei Fehler) zurück-

geben muß, zeigt das Ergebnis einfach nur auf einen Speicherbereich - ein „Beispiel"-Objekt wird daraus erst, wenn der entsprechende Konstruktor aufgerufen wird.

Bei einem Ausdruck wie

```
Beispiel *bp = new Beispiel;
```

ruft der Compiler zuerst die Funktion

```
Beispiel::operator new(sizeof(Beispiel))
```

auf und hofft, daß diese einen von 0 verschiedenen Zeiger zurückgibt. Das Ergebnis weist er dann an `"bp"` zu und ruft gegebenenfalls einen Konstruktor auf dem entsprechenden Speicherbereich auf. Ein selbstdefiniertes `"new"` und `"delete"` braucht sich also nur um die Speicherverwaltung zu kümmern, während die Sache mit den Konstruktoren und Destruktoren weiter Sache des Compilers bleibt.

Bestimmt sind Sie jetzt begierig, zu erfahren, was es oben mit dem zweiten `"new"`-Operator (der mit dem zusätzlichen Parameter) auf sich hat. Man darf unmittelbar hinter einem `"new"` eine Argumentliste angeben, die dann direkt als zusätzlicher Parameter (der erste Parameter ist nach wie vor die Elementgröße) an eine passende `"new"`-Funktion übergeben werden, z. B. so:

```
#include <stream.h>
#include <stddef.h>
#include <stdlib.h>

class C
{ int i, j, k;
  public:
     void *operator new(size_t, char*);
     void *operator new(size_t, int, int);
     C();
};

void *C::operator new(size_t Size, char *Message)
{ cout « Message « "\n";
  return ::new char[Size];  // So greift man gezielt
                                    // auf das globale "new" zu
}

void *C::operator new(size_t Size, int x, int y)
{ cout « "Mit den Argumenten "« x« " und "« y « " werden "
  « Size « " Bytes alloziert.\n";
  return ::operator new (Size);  // So geht's auch
}

C::C()
{ cout « "Objekt " « this « " wird initialisiert.\n";
  i = j = k = 0;
}
```

```
void main()
{ C *cp1 = new("And now to something completely different")
  C,   *cp2 = new(17,4)C;
}
```

Die beiden "new"-Anweisungen im Hauptprogramm übergeben zusätzliche Parameter an den "new"-Operator, so daß die entsprechenden Funktionen aus der Klasse "C" aufgerufen werden. Diese geben hier lediglich ein paar Meldungen aus und reservieren dann Speicher mit dem standardmäßigen "new"-Operator. Normalerweise dienen zusätzliche Argumente bei "new" dazu, dem Operator näher zu sagen, wie und wo er Speicher allozieren soll.

Ganz schlaue Programmierer werden sich jetzt fragen, warum ein "new" als ersten Parameter immer die Objektgröße haben muß, obwohl man diese Größe doch auch mit "sizeof(Klasse)" berechnen kann. Dafür gibt es gleich zwei gute Gründe:

(1) Die Operatoren "new" und "delete" werden vererbt:

```
#include <stddef.h>

class Basis
{ // ...
  public:
    void *operator new(size_t);
};

class Klasse : public Basis
{ // ...irgendwas
};

Klasse *kp = new Klasse;  // Benutzt "Basis::operator new"
```

Im allgemeinen ist eine abgeleitete Klasse größer als ihre Basisklasse, und deshalb benötigt ein "new"-Operator wirklich eine präzise Angabe über die Größe des zu reservierenden Speichers.

ABER: Wenn eine Klasse "C" einen eigenen "new"-Operator besitzt, wird dieser nicht benutzt, um einen Vektor von "C" einzurichten (z. B. "new C[50]"). Das klingt ziemlich komisch, aber es gehört zu den Prinzipien von C++, daß eine abgeleitete Klasse alle Eignschaften ihrer Basisklassen erbt und folglich auch deren Speicherverwaltung, oder anders ausgedrückt: Da im obigen Beispiel eine „Klasse" immer gleichzeitig ein „Basis" ist, macht es auch Sinn, das "new" von „Basis" auch für „Klasse" zu verwenden - vielleicht funktioniert die Basisklasse ja nicht richtig, wenn sie nicht mit ihrem eigenen "new" alloziert wurde. Auf dem Amiga kann man sich z. B. Klassen vorstellen, die irgendwelche Grafik- oder Sounddaten repräsentieren und deshalb in Chip-Memory liegen müssen, also zwangsläufig ein eigenes, selbstgeschriebenes "new" benötigen. Folglich müssen dann auch alle abgeleiteten Klassen im Chip-Ram liegen.

Andererseits ist ein "C[50]" ein Vektor, aber kein "C", und erbt damit im allgemeinen auch nicht die Eigenschaften von "C".

(2) Auch das globale `"new"`, das für alle möglichen Objekte, die kein eigenes `"new"` haben, benutzt wird, kann überladen werden, z. B. so:

```
#include <stddef.h>
#include <stream.h>

void *operator new(size_t S, void *Wo)
{ return Wo;
}

void main()
{ int i = 26731;

  int *ip = new(&i)int(42);

  cout « i; // Ausgabe: 42
}
```

Dieses trickreiche Programm definiert ein neues globales `"new"`, dem man sagen kann, wo es Speicher „reservieren" kann - oder sagen wir besser: an welcher bereits reservierten Adresse das neue Objekt liegen soll. Im Beispiel wird ein `"int"`-Objekt eingerichtet, indem die Adresse eines bereits (auf dem Stack) existierenden Objekts übergeben wird. Anschließend zeigt `"ip"` dann auf `"i"`, und `"i"` wird während dieses `"new"` erneut initialisiert.

Der „delete"-Operator

Kein `"new"` ohne `"delete"`, und so wird es Sie wohl nicht wundern, daß man auch den `"delete"`-Operator überladen kann.

Eine `"delete"`-Funktion kann nur innerhalb einer Klasse deklariert werden, und zwar höchstens ein `"delete"` pro Klasse (d. h. diese Funktion kann nicht überladen werden). Der erste Parameter muß ein `"void*"` sein und zeigt auf das zu löschende Element, ein optionaler zweiter Parameter vom Typ `"size_t"` enthält die Objektgröße. Der Ergebnistyp darf ausschließlich `"void"` sein, und genau wie `"new"` ist `"delete"` immer ein statischer Member und folglich auch niemals virtuell.

Zwei Beispiele:

```
#include <stddef.h>

class C
{ // ...
  public:
    void operator delete(void*);
};

class D
{ // ...
  public:
    void operator delete(void*, size_t);
};
```

Auch **"delete"** wird vererbt, d. h. wenn eine Basisklasse ein überladenes **"delete"** besitzt, wird dieses auch für die abgeleitete Klasse benutzt. Dabei gibt es aber (scheinbar) ein Problem, das am folgenden Beispiel hoffentlich deutlich wird:

```
#include <stddef.h>
#include <stream.h>

class Basis
{ int i;
  public:// "delete"-Operator:
    void operator delete(void*, size_t);

              // virtueller Destruktor:
    virtual ~Basis();  // Wichtig - siehe unten
};

void Basis::operator delete(void *Obj, size_t Size)
{ cout «    "Das Objekt " « Obj « " der Größe " « Size «
              " wird gelöscht.\n";
  ::delete Obj   // Globales "delete" benutzen
}

Basis::~Basis()
{ cout « "Basis-Objekt " « this « " wird destruiert.\n";
}

class Dummy // Dient nur dazu, der anderen
{ int dumm;
};            // Basisklasse einen Offset zu verpassen.

class Klasse : private Dummy, public Basis
{ int i, j, k;
  public:
    ~Klasse();
};

Klasse::~Klasse()
{ cout « "Klassen-Objekt " « this « " wird destruiert.\n";
}

void main()
{ Klasse *kp = new Klasse;
  Basis  *bp = kp;

  cout « "Adresse des Objekts:   " « kp « "\n";
  cout « "Objektgröße: " « sizeof(Klasse) « "\n";
  cout « "Adresse der Basisklasse: " « bp « "\n";
  cout « "Basisklassengröße:    " « sizeof(Basis) « "\n\n";

  // Jetzt wird's spannend:
  delete bp;
}
```

Wir haben hier also eine Klasse **"Klasse"** mit zwei Basisklassen. Aus der zweiten, „Basis", erbt **"Klasse"** einen **"delete"**-Operator. Naturgemäß ist die Basisklasse kleiner als die abgeleitete Klasse und hat auch, da sie in der Basisklassenliste an zweiter Stelle steht, einen Offset der Größe **"sizeof(Dummy)"**.

Das Programm erzeugt also mit dem normalen **"new"** ein Element der Klasse **"Klasse"** und weist den Zeiger an eine geeignete Variable zu. Außerdem wird ein Zeiber **"bp"** auf die Basisklasse gesetzt. Wie Sie sicher wissen, managet der C++-Compiler diese Typkonvertierung korrekt und addiert dabei den richtigen Offset, so daß die numerischen Werte von **"kp"** und **"bp"** nicht übereinstimmen.

Nun sagt das Programm, was es getan hat, und gibt Adresse und Größe sowohl des gesamten Objekts als auch der darin enthaltenen Basisklasse aus. Schließlich löscht es das Objekt - aber über den Basiszeiger **"bp"** und das selbstdefinierte **"delete"**!

Da der Destruktor von **"Basis"** als virtuell definiert ist, werden vor dem eigentlichen **"delete"**-Aufruf auch völlig korrekt erst die abgeleitete Klasse und dann die zweite Basisklasse destruiert (die erste „Dummy"-Basisklasse hat ja keinen Destruktor). So, und jetzt kommt's: **"Basis::operator delete"** wird aufgerufen, und das ist der Punkt, auf den ich hinaus will und weshalb ich dieses ganze Beispiel überhaupt vorexerziere:

In C++ gibt es keinen **"self"**-Pointer, d. h. wenn man einen Zeiger auf ein Objekt hat (das möglicherweise in Wirklichkeit nur ein Basisobjekt eines größeren Objekts ist), gibt es keine Möglichkeit, festzustellen, von welchem Typ das referierte Objekt wirklich ist und an welcher Speicheradresse es wirklich beginnt. Der Compiler hat bei **"delete bp"** also nur einen Zeiger auf die Basisklasse und müßte dem **"delete"**-Operator nun eigentlich diesen Zeiger auf die Basisklasse sowie deren Größe übergeben, und das wäre hier offensichtlich falsch.

Aber wie so oft in scheinbar ausweglosen Situationen bietet C++ eine elegante Lösung auch für dieses Problem: Wenn die Klasse **"Basis"** einen virtuellen Destruktor besitzt (so wie es oben der Fall ist), wird dort in der Tabelle der virtuellen Funktionen nicht nur ein Verweis auf den richtigen Destruktor, sondern auch eine Information über die wahre Objektgröße und den Offset der jeweiligen Basisklasse abgelegt. Im Beispiel stehen dort also der Offset (**"-sizeof(Dummy)"**), der von **"bp"** zu subtrahieren ist, und die tatsächliche Objektgröße (**"sizeof(Klasse)"**). Also gibt das Programm folgendes aus (wobei die Adressen natürlich andere Werte haben werden, wenn Sie das auf Ihrem Rechner ausprobieren):

```
Adresse des Objekts: 0x2b8d20
Objektgröße:  24
Adresse der Basisklasse:  0x2b8d24
Basisklassengröße: 8

Klassen-Objekt 0x2b8d20 wird destruiert.
Basis-Objekt  0x2b8d24 wird destruiert.
Das Objekt  0x2b8d20 der Größe 24 wird gelöscht.
```

Zum Vergleich: Ohne das kleine Wörtchen **"virtual"** wird folgendes ausgegeben:

```
Adresse des Objekts: 0x32a878
Objektgröße:  20
Adresse der Basisklasse:  0x32a87c
Basisklassengröße: 4

Basis-Objekt  0x32a87c wird destruiert.
Das Objekt  0x32a87c der Größe 4 wird gelöscht.
```

Das Gesamt-Objekt ist hier nur 20 statt 24 Bytes groß, weil der Zeiger auf die Virtual-Tabelle nun entfällt. Diese Speicherersparnis bezahlen wir aber damit, daß der Compiler beim `"delete bp"` nicht mehr weiß, was eigentlich los ist: Hier wird nur noch der Destruktor für `"Basis"` aufgerufen, und der `"delete"`-Operator erhält als Argumente lediglich einen Zeiger auf ein „Basis"-Objekt sowie dessen Größe. Unter Umständen kann die selbstdefinierte Speicherverwaltung damit ganz schön Probleme kriegen, denn ein solches Objekt wurde ja nie zuvor wirklich eingerichtet. Die Heap-Verwaltung von MaxonC++ ist aber robust genug um zu wissen, daß dieses Objekt Basis eines anderen ist.

4.2.2.3 Funktionsaufruf als Operator

Für Leute, die Pascal oder andere „harmlose" Programmiersprachen gewohnt sind, ist es ein gewöhnungsbedürftiger Gedanke, daß der Funktionsaufruf in C++ formal ein Operator ist, dessen linker Operand entweder ein (eventuell überladener) Funktionsname oder ein Ausdruck, der eine Funktion (in einigen Implementationen alternativ auch einen Zeiger darauf, siehe 2.3.2) darstellt, ist, während die Argumentliste als rechter Operand gilt.

Operanden sind, das haben Sie sicher schon gemerkt, zum Überladen da, und folglich kann auch der Funktionaufrufs-Operator, der durch `"()"` dargestellt wird, auf Klassen (also ausschließlich als Member-Funktion) überladen werden, z. B. so:

```
class Funny
{ int i;
  public:
    void operator()();
    int operator()(int, int);
};

#include <stream.h>

void Funny::operator()()
{ cout « "Hello, World!\n" « "i = " « i « "\n";
}

int Funny::operator()(int j, int k)
{ i = j+k;
  return i;
}

void main()
{ Funny f;
  int r = f(17,4);
```

```
   f();
}
```

Zunächst einmal sollten Sie sich nicht von den diversen Klammern, die sich hier anhäufen, irritieren lassen: erstens ist LISP viel schlimmer und zweitens ist `"()"` die Darstellung des Operators, `"operator()"` folglich ein Funktionsname und das Klammerpaar, das jeweils folgt, die Parameterliste - also genau wie bei jedem anderen Operator auch. Ebenso leicht ist die Funktion dieser Überladung zu erkennen: Wird ein Klassenobjekt mit einer Argumentliste wie eine Funktion aufgerufen, so ruft der Compiler die entsprechende Operatorfunktion wie eine ganz normale Memberfunktion auf.

Das folgende Programm ist deshalb zum obigen äquivalent, kommt aber ohne Überladen von `"()"` aus:

```
class LessFunny
{ int i;
  public:
    void operator_fun();
    int operator_fun(int, int);
};

#include <stream.h>

void LessFunny::operator_fun()
{ cout « "Hello, World!\n" « "i = " « i « "\n";
}

int LessFunny::operator_fun(int j, int k)
{ i = j+k;
  return i;
}

void main()
{ LessFunny f;
  int r = f.operator_fun(17,4);
  f.operator_fun();
}
```

Wie Sie sehen, dient das Überladen von `"()"` eigentlich nur dazu, daß man eine Memberfunktion aufrufen kann, ohne ihren Namen nennen zu müssen. Das impliziert auch schon die wichtigste Anwendung dieses Features: Den Operator `"()"` überlädt man auf Klassen, die in irgendeiner Weise Funktionen repräsentieren. Die Möglichkeit, `"()"` zu überladen, ist also alles andere als das zentrale Element der objekt-orientierten Programmierung, aber hin und wieder ist es ganz praktisch, weil es elegantere Programme ermöglicht.

4.2.2.4 Klassen als Möchtegern-Vektoren

Ähnlich wie der Funktiosnaufruf, ist auch der Indexoperator `"[]"` überladbar. `"operator[]"` muß dabei eine nicht-statische Memberfunktion sein.

Bekanntlich betrachtet C (und damit auch C++) die Vektor-Indizierung als binären Operator, und das ist bei überladener Klassen-Indizierung nicht anders. Dieses Rumgesülze soll ganz einfach zum Ausdruck bringen, daß man auch mit Überladen von "[]" keine echten mehrdimensionale Vektoren realisieren kann, weil der zweiter Operand auch hier nur ein einfacher Ausdruck sein darf:

```
class Vektor
{ int *daten;
  unsigned anzahl;
  public:
    int &operator[ ](unsigned); // Folgende Zeile geht NICHT:
    int &operator[ ](int, int); // Keine mehrdimensionalen
                                                  // Vektoren!
};

#include <stream.h>

int &Vektor::operator[ ](unsigned index)
{ if (index < anzahl)
    return daten[index];
  else
  { cout « "Murx!\n";
    exit(26731);
  }
}

void main()
{ Vektor v1;
  int i;

  // ...

  v1[2] = v1[i];
}
```

Rein formal entspricht der Ausdruck "a[b]" dem Funktionsaufruf "a.operator[](b)". Wegen dieser Semantik wird weder verlangt, daß der zweite Operand, also der „Index", irgendwie von einem numerischen Typ sein muß, noch will der Compiler als Ergebnistyp dieser Operation unbedingt eine Referenz (d. h. einen L-Wert) haben. Diese Typen treffen die gewohnte Semantik des Vektor-Zugriffs aber am besten, und so wird man beim Überladen von "[]" wohl stets über einen diskreten Datentypen indizieren und eine Referenz auf ein Objekt zurückgeben.

Den Operator "[ ]" wird man naheliegenderweise nur dann überladen, wenn eine Klasse irgendwie so etwas wie einen Vektor repräsentiert. Typische Beispiele sind Strings oder dynamische Vektoren, die sich automatisch dem benötigten Indexbereich anpassen. Man könnte "[]" aber z. B. auch auf der Telefonlisten-Struktur aus unserem kanonischen Beispielprogramm überladen, und zwar möglicherweise gleich mehrfach:

```
{ class Telefonbucheintrag
  public:char Name[40], Nummer[20];
};
```

```
class Telefonbuch  // Irgendwelche Basisklassen...
{ // usw.
  public:
    Telefonbucheintrag &operator[ ] (int nr);
    Telefonbucheintrag &operator[ ] (const char *name);
};

Telefonbuch T1, T2;

#include <stream.h>

void CrunchyFrog()
{ cout « T1[42].Name « T2["Monty Python"].Nummer;
}
```

In diesem Fall könnte die konventionelle Indizierung mit `"int nr"` eine Referenz auf das `"nr"`-te Listenelement liefern, während man bei der Indizierung mit einem String den Telefonbucheintrag mit diesem Namen heraussuchen könnte, was dann einem assoziativen Array entspräche.

Zeiger und Vektoren sind in C eng miteinander verwandt, und deshalb schließt sich das folgende Feature geradezu wunderbar an:

4.2.2.5 Intelligente Zeiger: "->" und das unäre "*"

Beim unären `"*"`-Operator ist zum Thema Überladung eigentlich nicht viel zu sagen, denn das geht genau wie bei `"+"` oder `"!"`:

```
// entweder als Member...
class C1
{ C1 operator * ();
};

// oder einfach so:
class C2 {  };

int operator *(C2);
```

Witziger ist da schon `"->"`, denn das ist eigentlich kein echter Operator: Normalerweise ist der linke Operand ein Zeiger und der rechte der Name eines Members, also alles andere als ein Audruck. Deshalb weicht die Semantik bei der Überladung von `"->"` etwas von den „normalen" Operatoren ab.

`"->"` kann nur als nicht-statische Memberfunktion deklariert werden und hat originellerweise auch keine weiteren Parameter, weshalb weiteres Überladen innerhalb einer Klasse hier flach fällt. Ein Ausdruck wie `"x->y"` wird, wenn `"x"` ein Klassenausdruck ist, als

```
(x.operator->())->y
```

ausgewertet. Folglich sollte der Ergebnistyp dieses Operators tunlichst etwas sein, worauf der Compiler dann `"->y"` auch anwenden kann, also normalerweise ein Zeiger auf eine Klasse, in der `"y"`

Member ist (alternativ könnte das Ergebnis natürlich auch eine weitere Klasse sein, auf der "->" überladen ist... - aber so schlimm muß man es ja nicht unbedingt treiben). Ein simples Beispiel:

```
struct Objekt   { int a, b, c; };

class Zeiger
{ int foo, bar;
  public:
    Objekt *operator ->();
    Objekt &operator *();
};

Objekt Foobar;

Objekt *Zeiger::operator->()
{ return &Foobar;
}

Objekt &Zeiger::operator*()
{ return Foobar;
}

void main()
{ Zeiger z1, z2;

  // Alle folgenden Operationen manipulieren das Objekt "Foobar":
  z1->a = 42;
  z2->b = 76;
  (*z2).c = 26731;
}
```

Zugegebenermaßen ist es nicht unbedingt sinnvoll, wenn "*" und "->" immer Verweise auf ein einziges Objekt liefern. Typische Anwendungen für solche Überladungen sind „intelligente Zeiger": Beispielsweise könnte man die gerade nicht benötigten Teile einer großen Datenstruktur auf Festplatte oder Diskette auslagern und Objekte möglichst nur dann ins RAM laden, wenn sie benutzt werden. Die Operatoren "*" und "->" müßten dann feststellen, ob das referierte Objekt sich gerade im Speicher befindet, und es andernfalls vom Massenspeicher laden.

4.2.2.6 Inkrement und Dekrement

Die Operatoren "++" und „--" sind außerordentlich bemerkenswert, weil sie als einzige unäre Operatoren gleichermaßen als Postfix- und als Präfixoperatoren benutzt werden können. In ihrer Standardbedeutung haben sie dabei sogar unterschiedliche Funktionen, denn "++c" und "c++" liefern bekanntlich unterschiedliche Ergebnisse. Das Problem ist nun, was der Compiler machen soll, wenn "++" oder "--" vom Anwender überladen werden.

Im ersten C++-Standard konnte man nur jeweils einen Operatoren definieren, der dann sowohl bei Postfix- als auch bei Präfixausdrücken aufgerufen wurde. Es gab für die Operatorfunktion keine Möglichkeit festzustellen, auf welche der beiden Arten sie benutzt wurde. Es stellte sich aber bald heraus, daß diese Konvention nicht zu befriedigen vermochte, denn schließlich dient der ganze

Schmonz mit den überladenen Operatoren hauptsächlich dazu, die gewohnten Funktionsweisen von Operatoren auf selbstdefinierten Datentypen zu imitieren. Also griff man bei C++ 2.0 zu einer anderen Lösung.

Werden die Operatoren `"++"` und `"--"` unär überladen, so bezieht sich das auf die Präfix-Schreibweise `"++x"`. Man kann diese Operatoren aber auch mit einem zweiten „Phantom-Operanden" des Typs `"int"` versehen, und das entspricht dann einem Postfix-Ausdruck:

```
class N
{ int n;
  public:
    N operator ++();
    N operator ++(int);
};

N n1;

void main()
{ ++n1;      // Entspricht n1.operator++()

  n1++;      // Entspricht n1.operator++(0)
}
```

Dem Phantom-Parameter wird also pauschal das Argument `"0"` übergeben, denn er dient ja ausschließlich dem Zweck, daß die beiden Funktionen sich unterscheiden sollen.

Bei global definierten Funktionen geht das völlig analog:

```
class M
{ public:int m;
};

M operator ++(M);
M operator ++(M, int);

M m1;

void main()
{ ++m1;      // entspricht operator++(m1);

  m1++;      // entspricht operator++(m1,0);
}
```

Es gibt eine furchtbar schlaue Begründung dafür, weshalb man das gerade so und nicht genau andersrum definiert hat, aber die ist mir gerade entfallen, und ich möchte Ihnen auch nicht zumuten, sich das zu merken. Im Zweifelsfall wissen Sie ja, wo Sie es nachschlagen können.

Ach ja, auch wenn hier nur Beispiele mit `"++"` stehen, funktioniert das Überladen von `"--"` natürlich ganz genauso.

4.2.2.7 Ein- und Ausgabe

So manche Daten, die irgendwie so anfallen, möchte man gern auch ausgeben. Am schönsten wäre es natürlich, wenn das ganz normal ginge, so mit `"cout"` und so...

Aber hallo, das geht, und wie! `"cout"` ist nichts anderes als ein bestimmtes Objekt der Klasse `"ostream"`, und der Operator `"«"`, der sonst eigentlich Bits durch die Gegend schiebt, ist mit dieser Klasse als linken Operanden mehrfach überladen. Klar, daß der mündige Programmierer das auch selbst kann.

Also, nehmen wir einmal an, Sie haben eine Klasse namens `"Complex"`, die rein zufällig auch komplexe Zahlen repräsentiert. Nun wäre es natürlich ganz nett, wenn man diese Daten ganz einfach nach `"cout"` schicken könnte. Das ist nach all dem, was Sie bisher wissen, wirklich kein Thema mehr:

```
// Erster Versuch!

#include <stream.h>   // da wird "ostream" definiert

class Complex
{ public:
    double re, im;
};

void operator « (ostream os, const Complex c)
{ os « c.re « "+" « c.im «"i";
}
```

Das Problem dabei: Wenn `"«"` ein `"void"` zurückgibt, wie kann man denn dann mehrere Ausdrücke hintereinander ausgeben, so wie bei `"cout « a « b"`? Also muß eine derartige Funktion auf jeden Fall ein Objekt der Klasse `"ostream"` zurückgeben. Da man aber nie so genau wissen kann, wie ein Objekt reagiert, wenn es kopiert wird, arbeitet man hier besser mit Referenzen, und zwar sowohl beim Funktionsergebnis als auch beim `"ostream"`-Parameter:

```
// So ist's besser:
#include <stream.h>

class Komplex
{ public:
    double re, im;
};

ostream &operator « (ostream &os, const Komplex &c)
{ os « c.re « "+" « c.im «"i";
  return os;
}

void main()
{ Komplex k;
  k.re = 1.0;
  k.im = 2.0;
```

```
   cout « k;
}
```

Da wir gerade so schön mit Referenzen hantieren, deklarieren wir im obigen Beispiel auch den „Komplex"-Parameter als Referenz, immerhin spart eine Referenz im Vergleich zu einem Class-Parameter normalerweise so einiges an Rechenzeit.

Wenn Sie wissen, daß `"cin"` ein Objekt der Klasse `"istream"` ist, dürfte Ihnen jetzt auch klar sein, wie man die Eingabe eines Objekts bewerkstelligt:

```
#include <stream.h>

class X
{ public:
     char x[50];
};

istream &operator » (istream &im, X &y)
{ im » y.x;
  return im;
}

void main()
{ X x1, x2;
  cin » x1 » x2;
}
```

Sie dürfen nur die `"return"`-Anweisung in solchen Operatorfunktionen niemals vergessen, denn sonst gibt es echt eklige und schwer zu findende Abstürze, aber holla!

4.2.3 Beispielprogramm: Die „string"-Bibliothek

Es dürfte allgemein bekannt sein, daß BASIC die leistungsfähigste und komfortabelste Programmiersprache ist.

Lacht da etwa jemand?

Na gut, leicht übertrieben ist das schon, aber eines muß man BASIC lassen: es gibt keine andere gebräuchliche Programmiersprache, die ein so müheloses und intuitives Stringkonzept besitzt. Man sagt ganz einfach `"A$"`, und der Interpreter weiß Bescheid und erledigt den Rest. Da kann höchstens noch Rexx mithalten, aber Rexx ist weder „gebräuchlich" noch eine „Programmiersprache".

In jeder „echten" Programmiersprache muß man zumindest bei der Deklaration einer Stringvariablen angeben, wie lang das Ding einmal werden soll, und das Stringhandling von C ist bekanntermaßen völlig greuslich, da faktisch eigentlich gar nicht von einem Stringhandling die Rede sein kann. Also sollte man doch besser auf BASIC umsteigen, oder wie?

Keine Angst, Sie haben das Geld, daß Sie für MaxonC++ hingeblättert haben, nicht umsonst verschleudert. Denn erstens hat MaxonC++ eine String-Bibliothek, die fast so komfortabel wie BASIC ist, und zweitens werde ich eine Mini-Version dieser Bibliothek hier im Quelltext vorstellen, denn daran lassen sich viele Konzepte, wie Copy-Konstruktoren, Überladen von Operatoren, Konvertierungsfunktionen oder temporäre Objekte anschaulich darstellen. Dieses Beispielprogramm soll der krönende Abschluß des Kapitels 4 werden.

Das mit dem **"$"**-Zeichen hinter dem Variablennamen können wir leider (?) nicht von BASIC abkupfern, denn man wird eine Stringvariable deklarieren müssen wie jede andere auch. Dafür dürfen Variablennamen auch mehr als zwei Zeichen lang sein (wie war das doch gleich im C64-BASIC?) und werden ganz einfach mit dem Datentyp **"String"** ohne jede Längenangabe deklariert. Außerdem werden unsere Strings lauter Dinge können, die mit den normalen C-Strings - oder sagen wir besser „Zeichenvektoren" - nicht so einfach gehen, nämlich Zuweisungen ohne **"strcpy"**, Vergleiche ohne **"strcmp"** und Aneinanderhängen ohne **"strcat"**.

Also, machen wir uns ans Werk. Sie können sich sicher schon denken, daß wir hier eine Klasse namens **"String"** deklarieren werden. Die enthält dann lediglich Informationen über den String, während die eigentliche Zeichenkette in einem dynamisch mit **"new"** reservierten Speicherbereich liegen wird. Auf diese Zeichenkette, die wie immer mit einem Nullbyte abgeschlossen wird, zeigt der private Member **"str"**. Wir vereinbaren dabei, daß der Wert 0 hier eine gültige Repräsentation für den leeren String darstellen soll. Weil es uns fast nichts kostet und an einigen Stellen nutzen wird, speichern wir auch die Stringlänge ab, und zwar im Member **"len"**. Da diese Daten zu den Eigenheiten der Implementierung gehören, ist es sinnvoll, sie als **"private"** zu deklarieren.

Also geht die Definition unserer Klasse so los:

```
class String
{  char *str;
   int len;
```

Natürlich wird unsere Klasse diverse Konstruktoren haben. Diese und einige anderen Funktionen werden aber etwas gemeinsam haben: Zu einer gegebenen Zeichenkette (ein **"char*"**, nicht etwa ein **"String"**) muß ein Speicherbereich geeigneter Größe mit **"new"** eingerichten, der String dorthin kopiert, ein Zeiger darauf in **"str"** eingetragen und die richtige Stringlänge in **"len"** vermerkt werden. Also deklarieren wir dafür eine (ebenfalls private) Memberfunktion:

```
void alloc (const char*);
```

mit folgender Definition:

```
void String::alloc (const char *initstr)
{  if (initstr)
   {  // Stringlänge feststellen, Speicher einrichten:
      len = strlen(initstr);
      str = new char[len+1];
      if (str)  // Neuen String initialisieren:
        strcpy(str, initstr);
      else         // "new" schlug fehl, also "aussteigen":
```

```
        exit(42);
    }
    else            // Initialisierung mit Leerstring:
    { len = 0;
      str = 0;
    }
}
```

Das Gegenstück ist die ebenfalls mehrfach benötigte Memberfunktion

```
void free();
```

die den für die Zeichenfolge reservierten Speicher freigibt und aus Gründen der Konsistenz den `"str"`-Zeiger wieder auf Null setzt:

```
void String::free()
{ if (str)
  { delete [ ] str;
    str = 0;
  }
}
```

Zur Erinnerung: Das `"[ ]"` hinter dem `"delete"` ist nötig, weil der Vektor mit variabler Größe eingerichtet wurde.

Zwei Objekte dürfen sich niemals dieselbe physikalische Zeichenkette, also einen allozierten Speicherbereich „teilen", indem die `"str"`-Zeiger beider Objekte auf dieselbe Adresse zeigen, denn dann kommt es fast zwangsläufig zu Kollisionen. Deshalb werden wir zur Initialisierung von Strings meist die Funktion `"alloc"` benutzen, die ja eine eigene Kopie ihres Arguments anlegt. Es gibt aber auch die Situation, daß eine Funktion bereits einen Zeichenvektor dynamisch alloziert und initialisiert hat und genau diese Zeichenkette als `"str"` in ein Stringobjekt eingetragen werden soll, ohne daß sie noch einmal kopiert werden sollte. Für diesen Zweck führen wir einen Konstruktor ein, der nichts anderes tut, als einen Stringzeiger und eine Länge einzutragen:

```
String (int, char*);
```

Da auch dies eine Low-Level-Operation ist, ist auch dieser Konstruktor privat. Seine Definition ist trivial und sieht folgendermaßen aus:

```
String::String (int len, char *str) : len(len), str(str) { }
```

Da gibt es also wirklich nichts mehr zu erklären - die Datenmember des Objekts werden mit bestimmten Werten initialisiert, und das ist es dann.

Kommen wir nun zu den öffentlichen Konstruktoren: Wir brauchen eigentlich deren drei, nämlich einen Default-Konstruktor, der ein „String"-Objekt mit dem Leerstring initialisiert, einen Konstruktor, der einen gewöhnlichen `"char[]"`-String in ein Objekt der Klasse `"String"` verwandelt und somit einen Teil der Schnittstelle zwischen unserer Stringbibliothek und der Standard-Stringverwaltung von C darstellt, und einen Kopier-Konstruktor, denn es dürfen ja niemals zwei String-Objekte

auf denselben Zeichenvektor verweisen, weshalb beim Kopieren von Objekten immer auch eine Kopie der Zeichenkette angelegt werden muß.

Mit Hilfe eines Default-Arguments können wir aus diesen drei aber zwei machen:

```
public:String (const char* = 0);
   String (const String &);
```

Auch deren Definition ist trivial, da wir ja schon die äußerst nützliche Funktion `"alloc"` haben:

```
String::String (const char *initstr)
{ alloc(initstr);
}

String::String (const String &s)
{ alloc (s.str);
}
```

Nun fehlt uns noch ein Destruktor:

```
~String();
```

Die Funktion `"free"` macht eigentlich schon alles, was der Destruktor leisten muß:

```
String::~String()
{ free();
}
```

Wir haben einen Konstruktor, der eine Zeichenfolge in einen String umwandelt. Da liegt es doch nahe, eine Konvertierungsfunktion zu schreiben, die gerade das Gegenteil macht. Zur Abwechslung deklarieren wir sie gleich als `"inline"`-Member innerhalb der Klassendefinition:

```
operator char*() const
{ if (str)
     return str;
  else
     return "";
}
```

Bei einer Wertzuweisung unter Strings müssen wir denselben Aufwand treiben wie beim Kopier-Konstruktor, nämlich eine Kopie des Zeichenvektors anlegen. Also ist auch noch der Operator `"="` zu überladen:

```
String &operator = (const String &);
```

Auch diese Memberfunktion hat eine vergleichsweise triviale Definition: Wir geben den alten Stringinhalt frei, tragen einen Zeiger auf den kopierten neuen Wert ein und geben eine Referenz auf das soeben initialisierte Objekt zurück:

```
String &String::operator = (const String &s)
{ free();
  alloc(s.str);
```

```
  return *this;
}
```

Fällt Ihnen etwas auf? Hier gibt es eigentlich nur ausgesprochen kurze Funktionen. Das ist so auch voll und ganz beabsichtigt, denn objektorientierte Programmierung soll ja dazu führen, daß die Algorithmen trivial werden, indem man die Datenstrukturen intelligent wählt.

Also geht es ebenso trivial weiter: Wir bieten dem Benutzer der Klasse noch einen schnellen und bequemen Zugriff auf die Stringlänge an, wieder in Form einer **"inline"**-Memberfunktion:

```
int length() const
{ return len;
}
```

Nun bleibt nur noch eins: Der Operator **"+"**, der zwei Strings aneinander hängen soll, muß auf intime Daten der Klasse zugreifen und wird deshalb als **"friend"** deklariert:

```
friend String operator + (const String &s1, const String &s2);
```

Da wir auf die Funktionen aus „*<string.h>*" zurückgreifen können, ist es nicht weiter aufwendig, diese Funktion zu implementieren:

```
String operator + (const String &s1, const String &s2)
{ // Wie lang ist das Ergebnis?
  int newlen = s1.length()+s2.length();

  // Speicher anfordern:
  char *newstr = new char[newlen+1];   // Wenn's geklappt
  // hat,dann String darin montieren:
  if (newstr)
  { strncpy (newstr,s1,s1.length());
    strncat (newstr,s2,s2.length())
  }

  // Mit dem "Spezialkonstruktor" wird aus Länge und Stringzeiger
  // ein Stringobjekt erzeugt:
  return String(newlen, newstr);
}
```

Spaßeshalber implementieren wir jetzt auch noch den Vergleichsoperator **"=="**, der zwei Strings mit **"strcmp"** auf zeichenweise Gleichheit testet. Diese Funktion muß weder als Member noch als **"friend"** deklariert werden, da sie auch so an alle für sie notwendigen Daten herankommt.

```
int operator == (const String &s1, const String &s2)
{ return !strcmp(s1,s2);
}
```

Der Aufruf von **"strcmp"** geht hier absolut locker und problemlos, da ja eine Konvertierungsfunktion von **"String"** nach **"char*"** existiert.

Man beachte, daß diese Funktion nicht die gewöhnliche Vergleichsoperation auf Zeigern und Vektoren überdeckt: Bei

```
char *s;   if (s == "Nanu") ...
```

werden jetzt immer noch die Adressen der beiden Zeichenketten verglichen. Unser selbstdefinierter Operator wird also nur aufgerufen, wenn mindestens ein Operand ein Objekt der Klasse `"String"` ist:

```
String t;   if (t == "Jau") ...
```

Natürlich kann man das auch durch ein explizites Casten erzwingen:

```
char *u;
if (String(u) == String("Jawoll!")) ...
```

Zu unserem Glück fehlt uns noch eine Möglichkeit, Strings auszugeben. Natürlich kann man das mit

```
String x;
cout « (char*)x;
```

bewerkstelligen, aber erstens kann man hier den Cast wegen der am Ende von 4.1.2.1 genannten Gründe nicht weglassen und zweitens ist diese Methode sehr unsicher, wenn der Compiler ein temporäres Objekt einführt, etwa bei

```
String x, y;
cout « (char*)(x+y);          // Vorsicht!
```

Deshalb überladen wir einfach den Operator `"«"`, so wie es in 4.2.2.7 gelernt haben:

```
ostream &operator « (ostream& out, const String& s)
{ return out « (char*)s;
}
```

Die eigentliche Ausgabe und die Rückgabe der Referenz in einer einzigen Anweisung zu verbinden, ist zwar trickreich, aber natürlich erlaubt, denn `"out « xxx"` gibt ja generell eine Referenz auf `"out"` zurück, und genau das soll auch unsere eigene Operatorfunktion als Ergebnis liefern.

Haben Sie die Übersicht verloren? Das kann ich gut verstehen, denn in den Listing-Fragmenten ging es oben doch manchmal etwas durcheinander. Deshalb kommt jetzt noch einmal das Ganze Programm im Überblick:

```
  // Stringbibliothek (verkleinerte Version von "tools/str.h")
  // *** Deklaration der Klassen und Funktionen:

class String
{ char *str;
  nt len;
  void alloc (const char*);
  void free();
  String (int, char*);
```

```
   public:String (const char* = 0);
   String (const String &);
   ~String();
   operator char*() const
   { if (str)
        return str;
     else
        return "";
   }
   String &operator = (const String &);
   int length() const
   { return len;
   }
   friend String operator + (const String &s1, const String &s2);
};

int operator == (const String &s1, const String &s2);

class ostream& operator « (ostream&, const String&);

// *** Implementierung:

#include <string.h>
#include <stream.h>

void String::alloc (const char *initstr)
{ if (initstr)
   { len = strlen(initstr);
     str = new char[len+1];
     if (str)
        strcpy(str, initstr);
     else
        exit(42);
   }
   else
   { len = 0;
     str = 0;
   }
}

void String::free()
{ if (str)
   { delete [ ] str;
     str = 0;
   }
}

String::String (int len, char *str) : len(len), str(str) {
}

String::String (const char *initstr)
{ alloc(initstr);
```

```
}

String::String (const String &s)
{ alloc (s.str);
}

String::~String()
{ free();
}

String operator + (const String &s1, const String &s2)
{ int newlen = s1.length()+s2.length();
  char *newstr = new char[newlen+1];
  if (newstr)
  { strncpy (newstr,s1,s1.length());
    strncat (newstr,s2,s2.length())
  }
  return String(newlen, newstr);
}

String &String::operator = (const String &s)
{ free();
  alloc(s.str);
  return *this;
}

int operator == (const String &s1, const String &s2)
{ return !strcmp(s1,s2);
}

ostream &operator « (ostream& out, const String& s)
{ return out « (char*)s;
}

// *** Test- und Beispielprogramm:

void main()
{ String s1, s2="I didn't expect the", s3;
  s1 = "Spanish Inquisition";
  s3 = s2 + " " + s1;

  char *awurx = "Vor"+"sicht"; // Unsichere Operation, da
                                // temporäres Objekt!
  cout « s3+".\n";
}
```

5. Der Preprozessor

5.1 Allgemeines

Ich weiß, es ist für Sie ebenso wie für mich eine Zumutung, aber trotzdem muß ich mich jetzt nach den ganzen tollen objektorientierten Features über etwas so primitives wie den Preprozessor auslassen. Wie so ziemlich alles in C, kann man mit dem Preprozessor furchtbar starke Sachen machen, aber sein Programm auch dermaßen verwirren, daß es niemand mehr versteht.

Theoretisch ist der Preprozessor ein eigenständiges Programm, das den Quelltext vor-verarbeitet, bevor der C- oder C++-Compiler ihn zu sehen bekommt. Das Ergebnis ist im Prinzip ein vereinfachter Quelltext, bei dem der Compiler sich nicht mehr um triviale Dinge zu kümmern braucht, insbesondere um das Einbinden von Includes, Definieren und Ersetzen von Macros, Weglassen vom Kommentaren und bedingte Übersetzung, also das Einfügen bzw. Weglassen von Quelltextteilen in Abhängigkeit von Bedingungen.

Preprozessor und Compiler arbeiten normalerweise (aber was ist in Zeiten wie diesen schon normal?) völlig unabhängig voneinander, und keinen interessiert, was der andere macht oder gemacht hat. Deshalb könnte man einen C-Preprozessor durchaus für die Vorverarbeitung von Pascal-Programmen oder irgendwelchen anderen Texten benutzen, und es soll tatsächlich Leute geben, die so etwas tun.

Der Preprozessor von MaxonC++ ist fest in den Compiler integriert. Dadurch ist es nicht möglich, einen Text nur durch den Preprozessor laufen zu lassen, ohne ihn zu compilieren. Der Vorteil dieses Verfahrens ist, daß der Compiler dadurch schneller wird und nirgendwo einen zusätzlichen Zwischencode ablegen muß. Wenn Sie andererseits wirklich das Bedürfnis verspüren sollten, einen Text vorverarbeiten zu lassen, ohne ihn durch den Compiler jagen zu wollen, muß ich Sie auf die PD-Serie verweisen. Der MaxonC++ Preprozessor ist integrer Bestandteil des MaxonC++ Compiler Systems, und daran ist nichts zu ändern.

Gesteuert wird der Preprozessor über Quelltextzeilen, die mit einem `"#"` anfangen. Dahinter hat normalerweise der Name eines Proprozessorbefehls zu stehen. Unser kleiner Freund befolgt dann diesen Befehl und löscht ihn dann natürlich rückstandlos aus dem Quelltext, so daß der Compiler normalerweise nichts davon mitbekommt. Zwei solcher Anweisungen wurden in diesem Handbuch bereits benutzt: `"#include"` zum Einbinden von Headerdateien mit Definitionen und `"#pragma"` zum Umschalten zwischen ANSI C- und C++-Modus (wobei `"pragma"` noch wesentlich mehr kann).

Jedenfalls muß das `"#"` immer an Zeilenanfang stehen, eventuell mit ein paar Leerzeichen davor, und die Preprozessoranweisung geht genau bis zum Zeilenende. Hier liegt der Preprozessor „schief" zum Compiler, denn in C und C++ ist es ja vollkommen belanglos, wo eine Zeile anfängt oder aufhört. Wesentlich konsistenter wäre es, wenn man Preprozessoranweisungen an beliebiger Stelle mit einem `"#"` beginnen und dann mit einem `";"` oder einem anderen geeigneten Zeichen abschließen könnte, aber man hat den Sprachstandard nun einmal anders definiert und mich fragt ja kei-

ner. Merken Sie sich also, daß der Preprozessor streng zeilenorientiert arbeitet und ich ihn deshalb nicht besonders mag.

Aber nun zu `"#define"`, einer der nützlichsten Preprozessorfunktionen überhaupt.

5.2 Was Sie schon immer über #define wissen wollten, aber nicht zu fragen wagten

5.2.1 Makrodefinition

Wie jeder alte Grieche weiß, heißt „makros" (mit Betonung auf der letzten Silbe) so viel wie „groß". Fragen Sie mich aber bitte nicht, was das mit den Makros zu tun hat, die Gegenstand dieses Abschnitts sind.

Nach einer Zeile wie

```
#define Name Weiss der Geier was!
```

ersetzt der Preprozessor überall im Quelltext den Bezeichner `"Name"` durch `"Weiss der Geier was!"`. Hinter dem `"#define"` hat als erstes der Makroname zu stehen, der stets ein Bezeichner ist.

Offensichtlich kann man dieses Feature benutzen, um häufig wiederkehrende Texte abzukürzen, aber auch, um das Aussehen des Quelltexts stark zu verändern - dazu später mehr. Im frühen C-Standard war dies außerdem die einzige Möglichkeit, Konstanten zu definieren, etwa so:

```
#define N 100

int Vektor[N];
```

Nach der `"define"`-Anweisung wird das Makro `"N"` überall durch die Zahl `"100"` ersetzt. Unschön ist dabei, daß hier keine Scoperegeln gelten, denn der Preprozessor hat bekanntlich nicht so ganz viel mit dem Compiler zu tun:

```
void f()
{
#define x 50
}

void g()
{
#define x "Hello, World!" // ERROR: "x" re-defineirt
}
```

Es gibt für solche Makronamen nur einen einzigen, globalen Gültigkeitsbereich. Ein Makro darf allerdings zweimal mit dem gleichen Wert definiert werden, wobei eine unterschiedliche Anzahl von Leerzeichen egal ist:

```
#define SUM a+b
#define SUM a + b
```

Außerdem kann man eine Makrodefinition löschen, und zwar mit der Preprozessor-Anweisung `"#undef"`:

```
#define GANZ int

 GANZ i1, i2;       // OK

#undef GANZ

GANZ i3, i4;    // ERROR
```

Ein „Token" ist ein einzelnes Quelltextsymbol, also ein Bezeichner (z. B. `"xyz"`), eine Konstante (numerisch oder String), ein reservieres Wort (wie `"int"` oder `"private"`), ein Operator (z. B. `"+"`) oder ein sonstiges Trennzeichen (`"["`, `";"`). Dagegen sind Kommentare, Leerzeichen oder Preprozessoranweisungen keine Token. Der Preprozessor wandelt den Quelltext in eine Folge von Token um, die der Compiler dann direkt verarbeiten kann. Die Ersetzung von Makros erfolgt in C streng tokenorientiert:

```
#define PLUS +

int i;

void f()
{ i PLUS= 1;
} // ERROR
```

ist ein Fehler, denn obwohl `"PLUS"` und `"="` direkt aufeinander folgen, ergeben sich daraus nach der Makroexpansion die beiden Token `"+"` und `"="` und nicht etwa das Token `"+="`.

Ich kann nicht oft genug wiederholen, daß der Preprozessor keine Ahnung von C hat und der Compiler nicht weiß, wie die Tokenfolge seiner Eingabe zustande gekommen ist. Also werden Bindungsregeln und der Vorrang von Operatoren total ignoriert, womit man ganz schön heftig auf die Schnauze fallen kann:

```
#include <stream.h>

#define N 17
#define M N+4

void main()
{ cout « 2*M;
}
```

Die Ausgabe ist hier **„38"** und nicht etwa **„42"**, denn da der Preprozessor gnadenlos Token ersetzt, wird `"2*M"` als `"2*17+4"` expandiert. Deshalb sollte man Makros, in denen Operatoren vorkommen, immer klammern:

```
#define M (N+4)
```

Stillschweigend habe ich Ihnen hier ein verschachteltes Makro untergejubelt. In der Tat darf man Makros benutzen, um andere zu definieren, so wie oben `"N"` in der Definition von `"M"` vorkommt. Dabei werden bei der Definition eines Makros darin enthaltene Makros nicht sofort, sondern erst bei der Benutzung des Makros expandiert:

```
#define N 100
#define M (2*N)
#undef N

int i = M;     // ERROR: Bezeichner "N" unbekannt
```

Da `"N"` undefiniert wird, ist es im folgenden ein ganz gewöhnlicher Bezeichner und wird vom Preprozessor auch als solcher an den Compiler weitergegeben, d. h. `"M"` wird hier wirklich als `"(2*N)"` und nicht als `"(2*100)"` expandiert.

Übrigens arbeitet der Preprozessor in mehreren Schritten, wobei das Entfernen vom Kommentaren logisch gesehen lange vor dem Ersetzen und Definieren von Makros kommt. Deshalb wird folgendes Makro nicht das Gewünschte leisten:

```
#define REM //

REM Dies soll ein Kommentar wie in BASIC sein,
REM ist aber leider falsch!
```

Hier werden zuallererst Kommentare, also auch der leere Komentar `"//"`, aus dem Quelltext gekickt, und erst dann stellt der Preprozessor fest, daß `"REM"` als Makro definiert wird - und zwar als leeres Makro. Also ersetzt er im nachfolgenden Quelltext `"REM"` wunschgemäß überall durch nichts, aber das, was jeweils danach in der Zeile steht, bleibt unverändert stehen und wird an den Compiler weitergegeben.

Es gibt übrigens eine Art Konvention, daß Makronamen immer aus Großbuchstaben bestehen sollten. Dadurch soll es etwas leichter werden, in einem Quelltext zu erkennen, was nun „echt" ist und was noch vom Preprozessor verändert werden wird.

Makros sind ein wunderbares Mittel, seinen Quelltext vollkommen undurchschaubar zu machen. Auf dem UseNet kursieren schon Tips für alle, die einen Programmierer ärgern wollen: Fügen Sie doch einfach bei einem „Freund" in eines seiner Includefiles, z. B. in `"<stream.h>"`, ein paar neue „Makros" ein, etwa so:

```
#define while if
#define double int
```

Das bedauernswerte Opfer dürfte ziemlich lange brauchen, um herauszufinden, warum ein Programm nicht so läuft, wie es soll...

5.2.2 Makros mit Parametern

Erheblich cooler sind allerdings Makros mit Parametern. Man deklariert sie, indem man unmittelbar hinter den Makronamen eine geklammerte Bezeichnerliste setzt, zum Beispiel wie im folgenden beispielhaften Beispiel:

```
#include <stream.h>

#define ABS(n) ((n)>0 ? +(n):-(n))
#define MAX(p1,p2) ((p1)>(p2) ? (p1) : (p2))

void main()
{ int i = ABS(-7), j = MAX(i+1,9);
  cout « i « " " « j « endl;
}
```

Beim Makroaufruf ist dann für jeden Parameter eine beliebige Tokenfolge anzugeben. Die Argumente sind durch Kommata zu trennen, wobei Kommas innerhalb von Klammern, wie das erste Komma im folgenden Beischlaf:

```
MAX(f(1,2), 42);
```

nicht als Trennung gelten, so daß obiger Makroaufruf korrekt ist.

Die einzige Anforderung an die Makro-Argumente ist, daß ihre Anzahl stimmen muß. Davon einmal abgesehen kann es sich um beliebige Tokenfolgen handeln, und als solche werden sie vom Prepro-zzessor auch behandelt. Deshalb ist es meist empfehlenswert, die Parameternamen in der Makrodefinition zu klammern:

```
#define DIFF(a,b) a-b

int i = DIFF(42, 17+4);
```

führt zu einem wahrscheinlich unerwarteten Ergebnis, denn der Aufruf von `"DIFF"` wird hier als `"42-17+4"` expandiert. Klüger ist deshalb

```
#define DIFF(a,b) (a)-(b)
```

Wenn man aber in der Makrodefinition ein `"#"` vor einen Parameternamen stellt, wird das zugehörige Argument als String-Literal expandiert:

```
#define TEST(string) cout « "Hello " « #string « "!";

#include <stream.h>

void main()
{ TEST(World)
}
```

Wird der Makroaufruf zu

```
cout « "Hello " « "World" « "!";
```

expandiert, d. h. der Preprozessor setzt gewissenmaßen Anführungszeichen um den Parameter.

Dagegen dient der Operator `"##"` zum Verbinden von Token:

```
#define MYIDENT(x) geheim_ ## x

int MYIDENT(i);
```

Durch das `"##"` wird der Makroaufruf nicht mit `"geheim_ i"`, sondern dem einzelnen Token `"geheim_i"` ersetzt. Dabei muß das Ergebnis der Verbindung aber immer ein einzelnes Token sein:

```
MYIDENT(!)
```

ist ein falscher Aufruf, denn `"geheim_"` und `"!"` bilden zusammen kein gültiges Token.

Makros mit Argumenten sind sehr beliebt, um solche Mini-Funktionen wie `"ABS"` oder `"MAX"` zu implementieren. Als C++-Programmierer sollte man sich aber überlegen, ob man für so etwas nicht doch lieber eine `"inline"`-Funktion benutzt, denn wenn der Compiler alle möglichen Optimierungen anwendet, ist das genauso effektiv.

5.3 Was Sie noch nie über #include wissen wollten, aber jetzt trotzdem erfahren

Die Preprozessoranweisung `"#include"` fügt den Inhalt einer Datei an der entsprechenden Stelle in den Quelltext ein. Als einziges Argument hat `"#include"` einen Dateinamen, wobei es zwei verschiedene Möglichkeiten gibt, diesen Dateinamen zu schreiben:

```
#include <name>
```

mit den spitzen Klammern sucht die Datei zuerst in einem ganz bestimmten Verzeichnis, wobei es verschiedene Möglichkeiten gibt, dieses Verzeichnis festzulegen:

- In der integrierten Entwicklungsumgebung von MaxonC++ legt man das Verzeichnis mit dem Menüpunkt **„Optionen/Compiler-Suchpfade"** fest, oder man stellt den gewünschten Pfad mit dem *„CPrefs"*-Programm ein.
- In der Kommandozeilenversion ist der Pfad *„MCPP:include"* als Default angegeben, den man mit der Option `"-i"` beliebig anders setzen kann. Damit kann man auch mehrere Pfade angeben, die der Preprozessor dann nacheinander nach der gewünschten Datei absucht.

Man benutzt diese Form von `"#include"`, um Standard-Includefiles einzubinden, z. B. *<stdio.h>* oder *<stream.h>*, aber auch die Amiga-Systemincludes.

Als kleines Bonbon vor allem für Diskettenbenutzer gibt es auch noch die Möglichkeit, die benutzten Dateien auf einen schnelleren Datenträger umkopieren zu lassen: Wenn Sie einen doppelten Suchpfad angeben, sucht der Preprozessor zuerst unter dem ersten Pfad, der z. B. in der RAM-Disk

liegen könnte. Findet er die Datei dort nicht, versucht er es über den zweiten Pfad, z. B. auf einer Diskette, und kopiert die Datei in das erste Verzeichnis um. Beim nächsten Übersetzungsvorgang liegt die Datei dann auf dem schnellen Medium vor. Übrigens legt der Preprozessor vor dem Umkopieren bei Bedarf selbständig die benötigten Unterverzeichnisse an.

Bei der zweiten Möglichkeit, ein Includefile einzubinden, setzt man den Dateinamen wie einen ganz gewöhnlichen String in Anführungszeichen:

```
#include "name"
```

Nun sucht der Preprozessor die Datei im aktuellen Verzeichnis. Diese Variante dient hauptsächlich dazu, selbstgeschriebene Dateien einzufügen. Der für Standard-Includes vorgesehene Pfad wird dabei zunächst nicht beachtet und folglich die Datei auch nicht umkopiert.

Unter MaxonC++ sind die beiden Varianten von `"#include"` allerdings praktisch gleichwertig, denn bei der `<xxx>`-Syntax versucht der Preprozessor es zuerst über den Standard-Pfad und bei Mißerfolg dann auch noch über das aktuelle Verzeichnis, und bei `"xxx"` ist die Reihenfolge gerade umgekehrt.

5.4 Bedingte Übersetzung

Mit dem Preprozessor kann man zur Übersetzungszeit Entscheidungen treffen, nämlich Teile des Quelltexts aufgrund bestimmter Bedingungen aus dem Quelltext auszublenden. Dazu dienen Konstrukte der Form

```
#if Bedingung

#endif
```

Ist die „Bedingung" erfüllt (was in C ja immer „ungleich Null" heißt), wird der nachfolgende Text gelesen, andernfalls wird alles bis zum `"#endif"` übersprungen.

Die Bedingung ist hier ein konstanter Ausdruck, der ganzzahlig ausgewertet wird. Dieser Ausdruck kann aus numerischen Konstanten und allen damit sinnvollen Operatoren bestehen. Makros werden im Ausdruck expandiert und alle anderen Bezeichner durch die Konstante „0" ersetzt. Außerdem können Ausdrucke wie

```
defined Name
```

oder

```
defined(Name)
```

darin auftauchen. Das Ergebnis ist `"1"`, wenn `"Name"` als Makro definiert ist, und andernfalls `"0"`.

Eine Zeile

```
#if defined x
```

kann ersetzt werden durch

```
#ifdef x
```

und entsprechend

```
#if !defined y
```

durch

```
#ifndef y
```

Die Abfrage, ob ein Symbol definiert ist, ist die wohl häufigste Form der bedingten Übersetzung. Beispielsweise geht es bei Include-Dateien oft ziemlich durcheinander, so daß es nicht immer ganz einfach ist, sicherzustellen, daß jede Datei nur einmal gelesen wird. Solche Probleme löst man normalerweise, indem man in jeder Datei ein bestimmtes Symbol definiert, anhand dessen der Preprozessor feststellen kann, ob er diese Datei schon gelesen hat:

```
#ifndef GANZ_SPEZIELLER_NAME
#define GANZ_SPEZIELLER_NAME

// Hier steht der eigentliche Inhalt der Datei.

#endif
```

Die Amiga-Systemincludes machen von diesem Schema ausgiebig Gebrauch.

Fast wie im richtigen Leben gibt es auch zu `"#if"` ein `"#else"`:

```
#if Bedingung

// Text1

#else

// Text2

#endif
```

Die Bedeutung dieses Konstrukts dürfte klar sein: Abhängig von der Bedingung wird entweder der erste oder der zweite Text gelesen und der jeweils andere übersprungen.

Ein Äquivalent zum `"switch"` hat der Preprozessor nicht, aber er erleichtert die Konstruktion von Mehrfachabfragen durch `"#elif"`, was sich natürlich von `"else if"` ableitet

```
#define N 2

#if N == 0     Text #0
#elif N == 1     Text #1
#elif N == 2     Text #2
#else     Text #3
#endif
```

Der `"else"`-Teil ist bei einer `"if"-"elif"`-Konstruktion natürlich optional. Gibt es kein `"#else"` und ist keine der Bedingungen wahr, wird eben das Ganze Konstrukt übersprungen.

Abschließend soll noch darauf hingewiesen werden, daß man solche Abfragen auch ineinander verschachteln kann. Gäbe es z. B. kein `"#elif"`, müßte man für obige Konstruktion folgendes schreiben:

```
#define N 2

#if N == 0     Text #0
#else
#if N == 1      Text #1
#else
#if N == 2       Text #2
#else
Text #3
#endif
#endif
#endif
```

5.5 Pragmatismus

5.5.1 Allgemeines und Wiederholungen

Bei jedem Standard kommt irgendwann der Punkt, wo er nicht mehr ausreicht. Deshalb ist im ANSI-Standard eine Möglichkeit vorgesehen, den Sprachumfang in einer Implementierung zu erweitern, ohne daß es dadurch zu Inkompatiblitäten zwischen den verschiedenen Compilern kommt: Die Preprozessor-Anweisung „#pragma".

Jede Implementierung kann selbst festlegen, was in einer Pragma-Zeile stehen darf und was es bedeuten soll. Es wird nur verlangt, daß eine unbekannte Syntax - also z. B. Pragmas eines anderen Compilers - nicht als Fehler gemeldet, sondern ignoriert werden muß. Das hat leider auch die Folge, daß der Preprozessor versehentlich gemachte Fehler nicht als solche meldet und man sich nachher vielleicht wundert, warum ein `#pragma` keine Wirkung hat.

Ein paar `#pragma`-Anweisungen wurden in diesem Handbuch schon an früherer Stelle vorgestellt, aber der Vollständigkeit halber seien sie hier noch einmal vorgestellt:

```
#pragma -
```

schaltet den Compiler in den ANSI C-Modus.

```
#pragma +
```

geht wieder zurück in den C++-Modus.

```
#pragma chip
```

legt alle nachfolgend deklarierten statischen Daten im CHIP-RAM ab.

```
#pragma fast
```

schaltet wieder in den normalen Modus zurück, d. h. alle Daten werden da abgelegt, wo gerade Platz ist.

5.5.2 Breakpoint-Kontrolle

MaxonC++ kann Programme erzeugen, die vom Benutzer abgebrochen werden können. Das ist einerseits während der Programmentwicklung angenehm, wenn man versehentlich eine Endlosschleife programmiert hat, und andererseits ist es oft sinnvoll, wenn der „normale" Anwender eines Programms die Programmausführung mehr oder weniger gewaltsam abbrechen kann.

Ein Programm fängt einen Abbruch ab, indem es an bestimmten Stellen, z. B. mindestens einmal innerhalb jeder Schleife, das Signalbit prüft, das durch die Tastenkombination <Ctrl>+<C> oder das CLI-Kommando `"break"` gesetzt wird. Natürlich kosten diese Tests Rechenzeit, einmal ganz zu schweigen von den dafür nötigen zusätzlichen Anweisungen. Außerdem ist es oft gar nicht wünschenswert, wenn eine bestimmte Aktion einfach so abgebrochen wird - man stelle sich einmal eine Textverarbeitung vor, die bei der Tastenkombination `"^C"` einfach aussteigt, ohne vorher abzuspeichern... Lange Rede, schwacher Sinn: Natürlich ist diese Abbruchmöglichkeit optional. Sie können Sie aktivieren, indem Sie die entsprechende Option im Optionen-Dialogfenster bzw. die Compileroption `"-gb"` in der Kommandozeilenversion benutzen, andernfalls ist sie deaktiviert.

Aber auch das ist in der Praxis noch nicht der Weisheit letzter Schluß: Vielleicht möchte man ja in einigen Programmteilen die Abbruchmöglichkeit benutzen und in anderen nicht. Dafür gibt es in MaxonC++ das Pragma `"break"`.

```
#pragma break +
```

schaltet das Setzen der Breakpoints ein und

```
#pragma break -
```

wieder aus.

Ein Beispiel:

```
void main()
{ int i, j=0;

#pragma break-

   for(i=0; i<100000; ++i)
     j = i-j;

#pragma break+

   for(i=0; i<100000; ++i)
```

```
    j = i-j;
}
```

In der ersten `"for"`-Schleife kann das Programm nicht abgebrochen werden, in der zweiten aber schon. Drückt man `"^C"`, während noch die erste Schleife abgearbeitet wird, geschieht zunächst einmal nichts. Erst zu Beginn der zweiten Schleife schaut das Programm wieder nach, ob das fragliche Signal eingetroffen ist, und steigt mit der Meldung

***** User break**

aus. Der Abbruch entspricht dabei einem `"exit(900)"`. d. h. es werden Dateien geschlossen und Speicher und sonstige Resourcen freigegeben. Auch mit `"atexit"` eingehängte Funktionen werden noch ausgeführt. Natürlich ist ein solcher Abbruch ein typisches Low-Level-Konstrukt, so daß dabei keine Destruktoren aufgerufen werden.

Ähnlich wie bei `"for"` werden auch bei `"while"`- und `"do"`-Schleifen sowie bei Schleifenverdächtigen `"goto"`-Sprüngen solche Abbruchtests eingefügt. Schlauberger werden jetzt natürlich wissen wollen, was bei

```
#pragma break+
while(Bedingung)
{ #pragma break-
}
```

ist, ob also der Wert zählt, der beim Schleifenanfang eingestellt ist, oder der, der beim Schleifenende gilt. Dazu kann ich nur sagen, daß noch nicht einmal ich das auswendig weiß und so etwas sich auch von Version zu Version ändern kann. Wenn man also „pragma break" benutzt, sollte man das so tun, daß es eindeutig ist.

Bleibt noch die Frage offen, was eine Abbruchkontrolle unter MaxonC^{++} nun wirklich kostet. Sie ist wohl aufs Äußerste optimiert und sieht in Assembler so aus:

```
move.l  Signaladresse, An
btst    #Signalnummer, (An)
bne     Ausstieg
```

Diese drei Instruktionen machen sich wirklich nur in sehr engen Schleifen bemerkbar. Die Laufzeit einer leeren `"for"`-Schleife wird dadurch z. B. noch nicht einmal verdoppelt, und bei Schleifen, die wirklich etwas tun (also in der Praxis so ziemlich alle) ist der relative Zeitverlust entsprechend gering.

5.5.3 Amiga-spezifische Systemaufrufe

In Abschnitt 2.2.1.2.4 wurde dargestellt, wie Routinen des Amiga-Betriebssystems aufgerufen werden, und außerdem wurde Ihnen versprochen, daß es außer der dort beschriebenen Pseudo-Link-Spezifikation noch eine andere Notation gibt, mit der man Systemfunktionen deklarieren kann. Voila, hier ist sie:

Wie Sie sich vielleicht denken können, hat auch dies mit einem **"pragma"** zu tun, und zwar mit dem pragma **"amicall"**. Eine solche Deklaration besteht im Wesentlichen aus vier Teilen:

1. Der Name der Basisvariablen
2. Der Offset als positive Zahl (d. h. der Compiler ergänzt sich das Minuszeichen selbst).
3. Der Funktionsname. Die Funktion muß bereits deklariert worden sein und darf aus Gründen der Eindeutigkeit nicht überladen sein.
4. Die Parameterliste, vertreten durch eine entsprechende Anzahl von Registernamen in runden Klammern.

Um das Ganze setzen wir ein Paar runder Klammern und noch **"#pragma amicall"** davor, und fertig ist die Deklaration.

Ein Beispiel:

```
#pragma amicall(DiskfontBase, 30, OpenDiskFont(a0))
#pragma amicall(DiskfontBase, 36, AvailFonts(a0,d0,d1))
#pragma amicall(DiskfontBase, 42, NewFontContents(a0,a1))
#pragma amicall(DiskfontBase, 48, DisposeFontContents(a1))
#pragma amicall(DiskfontBase, 54, NewScaledDiskFont(a0,a1))
```

Auch hier sei noch erwähnt, daß Sie solche Deklarationen normalerweise nicht selbst werden schreiben müssen, denn Sie bekommen sie für alle Amiga-Libraries mitgeliefert.

Nun werden sich sicher einige beschweren, daß MaxonC++ hier zwei unterschiedliche Konzepte für dieselbe Sache hat. Das ist natürlich richtig, aber beide Varianten haben ihre Vorteile:

- Die Link-Spezifikation ist in die Syntax der Sprache selbst integriert, was formal schöner als eine von außen „aufgepfropfte" Preprozessoranweisung ist. Außerdem eignet sich diese Variante auch für überladene Funktionsnamen, was zugegebenermaßen bei Betriebssystemfunktionen eher selten bis gar nicht vorkommt.
- **"pragma amicall"** ist der de-facto-Standard, für viele Amiga-Programmierer sogar das Pragma an sich. Auf Messen fragt z. B. so mancher „Kann der Compiler auch Pragmas?" und meint damit, ob man Betriebssystemfunktionen mit den üblichen Pragma-Zeilen direkt anspringen kann. Außerdem kollidieren Pragmas nicht mit dem C-Standard, da hier jeder Compiler alles kommentarlos überlesen muß, was er nicht kennt.

Bei den Include-Files, die mit MaxonC++ geliefert werden, hat **"#pragma amicall"** übrigens das Rennen gemacht, während die **"extern Amiga"**-Deklarationen in der vorliegenden Version nicht mehr auftauchen.

5.6 Noch mehr Features

5.6.1 Verbinden von Zeilen

Bekanntlich ist es in C absolut Banane, wo Zeilen anfangen oder aufhören. Dagegen ist dies beim Preprozessor durchaus relevant, denn hier muß jede Anweisung genau eine Zeile umfassen. Natürlich kann es dazu kommen, daß eine Zeile, z. B. durch eine umfangreiche Makrodefinition, arg lang und unhandlich wird. Für solche Gelegenheiten gibt es den Backslash („Gegenschrägstrich"). Wenn ein `"\"` am Zeilenende steht, wird diese Zeile mit der nachfolgenden verbunden, z. B. so:

```
#define Eumel \
26731
```

Dies ist eine gültige Makrodefinition, denn „26731" wird hier in die vorhergehende Zeile gezogen. Dieses Verfahren läßt sich fortsetzen, so daß man beliebig viele Zeilen logisch zu einer einzigen verbinden kann.

5.6.2 Verbinden konstanter Zeichenketten

Wenn im Quelltext mehrere Zeichenketten unmittelbar aufeinander folgen, macht der Preprozessor eine einzelne daraus:

```
char *s = "Dritte Tür links, " "jeder nur ein Kreuz!";
```

Man benutzt dieses Feature vor allem dann, wenn ein Teil einer konstanten Zeichenkette in Form eines Makros vorliegt, z. B.

```
#define DEVICE "dh1:"
#define SUBDIR "cpp"
#define FILE "M+Plus"

char Dateiname[] = DEVICE SUBDIR "/" FILE;
```

Dies ist absolut äquivalent zu und identisch mit

```
char Dateiname[ ] = "DH1:cpp/M+Plus";
```

Dieses Verbinden von Zeichenketten ist, wie Sie hier sehen, der so ziemlich letzte Arbeitsschritt des Preprozessors und erfolgt somit nach dem Ersetzen von Makros.

5.6.3 Trigraph-Sequenzen

Angeblich gibt es immer noch Rechner, die keinen vollständigen ASCII-Zeichensatz besitzen. Mitr persönlich fallen da zwar nur der C64 und diverse CP/M-Kisten ein, aber trotzdem hat die ANSI-Kommision auch an solche Phantomrechner gedacht und eine Möglichkeit eingeführt, auch auf solchen Systemen standardkonform C zu implementieren: die Drei-Zeichen-Folgen oder Trigraph-Sequenzen, wie der Lateiner sagt.

Der Preprozessor ersetzt einige Zeichenfolgen, die jeweils mit zwei Fragezeichen beginnen, durch ein anderes Zeichen, das in C bzw. C++ gebraucht wird. Im einzelnen werden folgende Ersetzungen vorgenommen:

```
??=  #
??(  [
??)  ]
??/  \
??<  {
??>  }
??'  ^
??!  |
??-  ~
```

Das folgende Programm illustriert dieses Feature:

```
??=include ??/     <stream.h>

void main()
   ??< char c??(??)="Test";
   cout << c;    ??>
```

Ausgesprochen hübsch, nicht wahr? Da diese Ersetzungen ganz zu Anfang der Vorverarbeitung erfolgen, kann man auch Überraschungen erleben:

```
#include <stream.h>

void main()
{  cout << "(Nanu???)";
}
```

Dieses Programm gibt

(Nanu?]

aus, denn die entsprechende Trigraph-Sequenz wird rücksichtslos durch die Klammer `"]"` ersetzt. Sollte man wirklich einmal in eine solche Verlegenheit kommen, kann man sie aber umgehen, indem man das oben erwähnte Verbinden von Zeichenketten benutzt, denn das geschieht erst lange nach dem Ersetzen der Drei-Zeichen-Folgen:

```
#include <stream.h>

void main()
{  cout << "(Nanu??" "?)";
}
```

5.6.4 Hausgemachte Fehlermeldungen

Die meisten Programmierer ärgern sich, wenn der Compiler einen Fehler meldet. Es gibt aber auch solche, die das dringende Bedürfnis haben, zusätzliche Fehlermeldungen selbst zu erzeugen, und die können sich freuen, denn C bietet für jeden etwas.

Eine Zeile der Form

```
#error Irgendwas
```

veranlaßt den Preprozessor, eine entsprechende Fehlermeldung auszugeben. Typische Anwendungen sind Tests, ob Makros definiert sind, etwa so:

```
#ifndef SYMBOL
#error SYMBOL ist nicht definiert!
#define SYMBOL 42  // Defaultwert
#endif

char c[SYMBOL];
int i = SYMBOL-1;
```

5.6.5 Getürkte Dateinamen und Zeilennummern

Tatsächlich werden nicht alle Programme vom Menschen geschrieben. Gerade C-Quelltexte werden oft von anderen Programmen erzeugt: Da gibt es Konvertierungsprogramme (im weitesten Sinne Compiler), die Pascal oder FORTRAN nach C umwandeln, nicht zu vergessen diese seltsamen C++-Frontends. Andere Beispiele sind Parsergeneratoren (ich bin übrigens stolz darauf, daß MaxonC++ vollständig ohne solche entstand - darum ist es ja auch so schnell), Oberflächengeneratoren wie MAXON's hervorragendes MakeAPP und andere Arten von CASE-Tools. Nehmen wir einmal an, wir haben ein Programm, in das auf der einen Seite eine Textdatei (also ein Quelltext im weitesten Sinn) hereinkommt und auf der anderen Seite ein C-Quelltext als Ausgabe erzeugt. Nun kann es natürlich sein, daß dieses C-Programm Fehler enthält, die auf eine fehlerhafte Eingabe zurückzuführen sind und deshalb vom Benutzer in der Eingabedatei korrigiert werden müssen. Dann wäre es natürlich hilfreich, wenn der C-Compiler tatsächlich die Fehlerposition in der Quelldatei melden würde. Oder man möchte das resultierende Programm gern mit einem Source-Level-Debugger testen, und dann möchte man natürlich nicht den C-Quelltext sehen (den man ja nicht selbst geschrieben hat), sondern direkt in der ursprünglichen Quelldatei debuggen.

Deshalb gibt es die Preprozessor-Anweisung `"line"`, und zwar in zwei verschiedenen Formen.

```
#line 42
```

sagt dem Compiler, daß die folgende Zeile bitteschön die Nummer 42 haben soll, und

```
#line 4711 "Suck"
```

gibt die Pseudo-Zeilennummer `4711` und den Pseudo-Dateinamen `"Suck"` vor.

In den nachfolgenden Zeilen wird die so vorgegebene Zeilennummer ganz gewöhnlich fortgezählt. Auch der Pseudo-Dateiname bleibt erhalten, bis ein neuer angegeben wird.

5.6.6 Nix dahinter

Eine leere Preprozessor-Anweisung der Form

```
#
```

ist auch erlaubt und hat keine Bedeutung.

5.7 Vordefinierte Symbole

Der Preprozessor besitzt einige nützliche vordefinierte Makros. Dabei handelt es sich zum Teil um ANSI C-genormte Symbole, andere sind Bestandteil von C++ oder spezielle Eigenheiten von MaxonC++. Es ist nicht möglich, diese Namen umzudefinieren.

```
__LINE__
```

Das Makro `"__LINE__"` liefert jeweils die Zeilennummer, in der es benutzt wird, als dezimale „int"-Konstante. Entsprechend enthält

```
__FILE__
```

den Namen der aktuellen Quelltextdatei in Form eines Strings, z. B. so:

```
#include <stream.h>

void main()
{ cout « "Dies steht in Zeile " « __LINE__
  « " in der Datei " __FILE__ ".\n";
}
```

Man beachte, daß der Wert des Makros `"__FILE__"` eine konstante Zeichenkette ist und als solche natürlich auch mit davor oder dahinter stehenden Zeichenketten verbunden wird, so daß oben keinesweges zwei `"«"` vergessen wurden.

Praktisch sind auch die beiden Makros

```
__DATE__
```

und

```
__TIME__
```

die das Datum bzw. die Uhrzeit der Übersetzung angeben, und zwar vor allem dann, wenn man ein Programm mit einer eindeutigen Versionsangabe versehen will:

```
#include <stream.h>
```

```
void main()
{ cout « "Version 08.15/4711 vom " __DATE__ ", " __TIME__ " Uhr\n";
}
```

Das Datum wird dabei in der für uns Europäer etwas ungewohnten Form „Monat Tag Jahr", z. B. „Mar 08 1992", geliefert, während die Uhrzeit ganz normal in der Form „Stunden:Minuten:Sekunden" vorliegt.

Das Makro

```
__STDC__
```

liefert in allen zu ANSI C kompatiblen Implementierungen den numerischen Wert 1 und soll ermöglichen, festzustellen, ob ein Compiler diesem Standard genügt, denn andernfalls ist `"__STDC__"` nicht definiert. Eine Anwendung:

```
#ifndef __STDC__
#error Tut mir leid, dieses Programm braucht einen ANSI C Compiler
#endif
```

MaxonC++ nimmt sich die Freiheit, `"__STDC__"` sowohl im C- als auch im C++-Modus zu definieren, denn im wesentlichen ist C++ zu ANSI C abwärtskompatibel. Wie Sie sich denken können, ist `"__STDC__"` wie auch die vorhergehenden Makros Bestandteil des ANSI C-Standards, im Gegensatz zum folgenden:

```
__cplusplus
```

ist als leeres Makro definiert, und zwar nur bei C++-Compilern. Folglich ist es in MaxonC++ im ANSI-Modus unsichtbar, so daß man mit

```
#ifdef __cplusplus
```

leicht den aktuellen Modus testen kann. Dazu dient auch das folgende Makro, das es aber nur in MaxonC++ gibt:

```
__COMPMODE__
```

ist in MaxonC++ immer definiert, und zwar im C-Modus mit der `"int"`-Konstanten 0 und unter C++ mit 1. Außerdem möchte man oft auch noch gerne testen, um welchen Compiler (und ggf. welche Version) es sich handelt. Deshalb definiert MaxonC++ zusätzlich ein Makro namens

```
__MAXON__
```

das immer das zehnfache der Versionsnummer (z. B. 10 für Version 1.0) als ganzzahlige Konstante liefert:

```
#if !defined __MAXON__ || __MAXON__ < 10
#error Dieser Quelltext wurde für MaxonC++ ab V 1.0 geschrieben.
#endif
```

Die folgende Tabelle faßt noch einmal die Namen aller vordefinierten Symbole zusammen, einschließlich ihrer Werte und des Standards, durch den sie definiert sind:

`__LINE__`	ANSI C	Zeilennummer (int)
`__FILE__`	ANSI C	Dateiname (konstanter String)
`__DATE__`	ANSI C	Datum „Mmm tt jjjj" (konstanter String)
`__TIME__`	ANSI C	Uhrzeit „hh:mm:ss" (konstanter String)
`__STDC__`	ANSI C	1 __cplusplus C++ 2.0 (kein Wert)
`__COMPMODE__`	MaxonC++	0 im C-Modus / 1 im C++-Modus
`__MAXON__`	MaxonC++	Versionsnummer, z. B. 10

5.8 Arbeitsweise des Preprozessors

Der Preprozessor von MaxonC++ arbeitet in einem einzigen Schritt, ohne irgendwelche Zwischenergebnisse abzuspeichern, und gibt seine Ausgabe auch direkt an den Compiler weiter. Trotzdem kann man sich den Preprozessor logisch als eine Reihe unabhängiger Programme vorstellen, die nacheinander den Quelltext bearbeiten und an das jeweils nachfolgende Programm weitergeben. Diese logischen Phasen der Vor-Verarbeitung sind im einzelnen die folgenden:

1. Trigraph-Sequenzen (5.6.3) werden ersetzt.
2. Endet eine Zeile mit einem Backslash `"\"`, wird sie mit der jeweils nachfolgenden Zeile verbunden (5.6.1).
3. Der Programmtext wird in eine Folge von Token zerlegt, wobei Kommentare durch Leerzeichen ersetzt werden.
4. Preprozessor-Anweisungen aller Art werden ausgeführt.
5. Makros werden expandiert.
6. Aufeinanderfolgende konstante Zeichenketten werden verbunden (5.6.2).

Diese Reihenfolge logisch streng voneinaner getrennter Operationen hat einige Folgen, die die Leistungsfähigkeit des Preprozessors zwar teilweise mindern, aber andererseits seine Arbeitsweise überschaubarer machen. Zum Beispiel ist folgendes nicht erlaubt, da die Ausführung von Preprozessor-Anweisungen logisch vor der Makro-Ersetzung erfolgt:

```
#define INC #include

INC <stdio.h>
```

Es ist möglich, Zeilen mit der Trigraph-Sequenz `"??/"` statt `"\"` zu verbinden, aber andererseits nicht möglich, `"\"` als Bestandteil einer Makrodefinition zu verwenden, und die Liste solcher Mög-

lichkeiten und Einschränkungen ließe sich noch beliebig fortsetzen. Ich hoffe aber, daß ungefähr klar geworden ist, was es mit dieser logischen Reihenfolge auf sich hat.

Damit wäre die C- und C++-Sprachbeschreibung auch schon komplett. Das folgende Kapitel gibt noch Informationen zu einigen Aspekten der Amiga-Programmierung unter MaxonC++.

6. Was Sie als Amiga-Programmierer wissen sollten

6.1 Die Schnittstelle zum Betriebssystem

Maxon C++ wird mit den jeweils aktuellsten Amiga-Includedateien geliefert. Daran mußten nur geringfügige Modifikationen vorgenommen werden, da in diesen Dateien an einigen wenigen Stellen z. B. die C++-Wortsymbole `"class"` oder `"template"` als Bezeichner benutzt werden. Maxon C++ unterstützt Aufrufe von Betriebssystemfunktionen wahlweise über direkte Einsprünge via `"#pragma"` oder über "glue functions". Da letztere in der voluminösen "amiga.lib" stehen, empfehle ich die erste Methode.

Als kleine Besonderheit ist es nicht nötig, die folgenden Librarieszu öffnen:

Name der Library	Basisvariable
exec.library	SysBase
dos.library	DOSBase
diskfont.library	DiskFontBase
graphics.library	GfxBase
intuition.library	IntuitionBase

Die Exec-Library ist sowieso immer offen, DOS wird vom Startup-Code für die Standardein- und -ausgabe geöffnet, und wenn "GfxBase", "IntuitionBase" oder "DiskFontBase" im Programm referiert werden, bindet Maxon C++ automatisch Code ein, der diese Bibliotheken öffnet und am Programmende wieder schließt. Natürlich schadet es auch nicht, wenn ein Programm beides zusätzlich selbst macht.

6.2 Start von der Workbench

6.2.1 Die Do-It-Yourself-Methode: die Funktion "wbmain"

Es gibt bekanntlich zwei verschiedene Arten, auf dem Amiga ein Programm zu starten, nämlich vom CLI (bzw. einer Shell) aus oder von der Workbench. Offensichtlich unterscheiden diese beiden Arten sich erheblich:

- Zu jeder Shell gehört ein Text-Fenster, in das ein Programm Ausgaben machen und aus dem es Eingaben lesen kann. Dieses Fenster entspricht in C++ der Standardein- und -ausgabe. Dagegen gibt es beim Start von der Workbench zunächst kein solches Fenster, so daß auch die Standard-I/O-Kanäle undefiniert sind.

- Beim Shell-Start werden dem Programm Argumente in der Kommandozeile übergeben und stehen über `"argc"` und `"argv"` direkt zur Verfügung, während ein von der Workbench gestartetes Programm Argumente in Form aktivierter Icons (und ihrer "Tool Types") besitzt.

Daraus folgt natürlich, daß beide Arten des Programmstarts ganz unterschiedlich gehandhabt werden müssen. Der Startup-Code von MaxonC++ prüft zunächst, ob das Programm von der Workbench oder vom CLI aufgerufen wurde. Beim CLI-Start springt es wie gewohnt in die Funktion `"main"` ein, während im Falle eines Starts von der Workbench eine Funktion namens `"wbmain"` aufgerufen wird.

Moment mal, Sie haben doch bisher nie eine Funktion dieses Namens definiert? Das macht nichts, denn wenn der Linker kein `"wbmain"` findet, nimmt er eben das vordefinierte aus den Libraries, und diese Funktion macht überhaupt nichts. Also terminiert ein Programm ohne eigenes `"wbmain"` sofort, wenn es von der Workbench gestartet wird.

Um ein Programm also von der Workbench aufrufbar zu machen, muß man eine eigene `"wbmain"`-Funktion deklarieren, und zwar im C-Linkmodus, da der Linker ein Symbol namens `"_wbmain"` erwartet. Als Parameter kann man optional einen Zeiger auf die Startup-Message (deren Typ `"struct WBStartup"` in *<workbench/startup.h>* definiert ist) deklarieren. In C sieht der Prototyp also folgendermaßen aus:

```
void wbmain(struct WBStartup *ws)
```

...und in C++ so:

```
extern "C" void wbmain(WBStartup *ws)
```

Diese Startup-Message enthält einige interessante Informationen, insbesondere den Zähler `"sm_NumArgs"`, der die Anzahl der aktivierten Icons angibt, und den Zeiger `"sm_ArgList"`, der auf einen entsprechend großen Vektor des Typs `"struct WBArg"` verweist.

Jedes `"WBArg"`-Element enthält den Namen der Datei (`"wa_Name"`) sowie ein Lock auf das Verzeichnis (`"wa_Lock"`), in dem die jeweilige Datei liegt. Um eine der Dateien zu öffnen, muß man also erst ihr Verzeichnis zum aktuellen Directory machen.

`"sm_NumArgs"` ist immer mindestens 1, und das erste (oder besser gesagt "nullte") Element des Vektors `"sm_ArgList"` repräsentiert Name und Verzeichnis des Programms selbst.

Die `"wbmain"`-Funktion muß (und darf) übrigens die Startup-Message nicht beantworten - das erledigt der Startup-Code von MaxonC++ am Programmende selbst.

Das folgende Programm demonstriert einen typischen Programmstart auf dem Amiga einschließlich Auswertung der Argumente:

```
#include <clib/exec_protos.h>
#include <clib/intuition_protos.h>
#include <workbench/startup.h>
#include <stdlib.h>
```

```
#define WW 640          // Fensterbreite
#define WH 200          // Fensterhöhe

NewWindow mynewwin =
 { 0, 0, WW, WH, 2, 1, CLOSEWINDOW,
   ACTIVATE | WINDOWSIZING | WINDOWDRAG | WINDOWDEPTH | WINDOWCLOSE,
   NULL, NULL, "Startup-Demo", NULL, NULL,
   100, 50, 640, 200, WBENCHSCREEN};

struct Window* mywin = 0;


void FensterOeffnen()
 // öffnet ein Fenster und bricht ab, falls das nicht klappt
 {
   mywin = OpenWindow(&mynewwin);
   if (!mywin) exit(1000);
 }


void WartenUndSchliessen()
 {
   // Auf Benutzer-Aktion, insbesondere Close-Gadget, warten:
   WaitPort(mywin->UserPort);

   // Fenster schlie·en:
   CloseWindow(mywin);
 }


void TextAusgabe(char *string, int x, int y)
 // String an Pixel-Position x/y ausgeben
 {
   IntuiText it = { 1, 0,          // FrontPen, BackPen
                    JAM1,          // DrawMode
                    0, 0,          // Offset x/y
                    NULL,          // Font
                    string,        // Text
                    NULL };        // Next
   PrintIText(mywin->RPort, &it, x, y);
 }


// ***** Shell-Start:

void main(int argc, char **argv)
 { int i;

   FensterOeffnen();

   TextAusgabe("Start vom CLI", 10, 20);
   for(i=0; i<=argc; i++)
```

```
    TextAusgabe( argv[i], 20, 30+9*i );

  WartenUndSchliessen();
 }


// ***** Start von der Workbench:

#ifdef __cplusplus
 #define LINKAGE_C extern "C"
#else
 #define LINKAGE_C
#endif

LINKAGE_C void wbmain(struct WBStartup *ws)
 { int i;

   FensterOeffnen();

   TextAusgabe("Start von der Workbench", 10, 20);
   for(i=0; i < ws->sm_NumArgs; ++i)
     TextAusgabe( ws->sm_ArgList[i].wa_Name, 20, 30+9*i );

   WartenUndSchliessen();
 }
```

6.2.2 Die einfache Methode: die Datei "wbstartup.h"

Es wurde eingewendet, daß es ziemlich mühsam ist, alles wie oben beschrieben selbst zu implementieren. Außerdem hat man auf diese Weise noch lange kein normales Textwindow für die Standard-Ein- und Ausgabe. Deshalb gibt es die Includedatei "*<wbstartup.h>*" und die Bibliotheksfunktion `"wbparse"`, durch die alles viel einfacher und sicherer wird.

Alles, was Sie tun müssen, ist, in eine Übersetzungeinheit Ihres Programms folgende Zeile aufzunehmen:

```
#include <wbstartup.h>
```

In jener Datei steht eine vollständige `"wbmain"`-Funktion, die wiederum eine Bibliotheksfunktion namens `"wbparse"` aufruft:

```
extern "C" void wbmain(struct WBStartup *w)
{ wbparse(w); }
```

Die Funktion `"wbparse"` macht nun folgendes: Die Workbench-Argumente aus der Startup-Message werden ausgewertet, und daraus wird ein Argumentstring-Vektor erzeugt. Mit diesen Argumenten wird dann in die `"main"`-Funktion des Programms eingesprungen. `"argv[0]"` zeigt dann also auf den Programmnamen und `"argv[1]"` bis `"argv[argc-1]"` auf die Dateinamen der aktivier-

ten Icons. Zuvor wird noch in das Verzeichnis gewechselt, in dem das erste Argument (also das Programm selbst) steht.

Wichtig

Wenn Sie "*wbparse*" bzw. "*wbstartup.h*" benutzen und in C++ programmieren, muß Ihr Hauptprogramm auch Argumente übernehmen.

```
main(int argc, char *argv[])
```

deklariert sein, sonst geht die Sache in die Hose (im C-Modus gibt es da keinen Unterschied).

Nun wäre es ganz angenehm, wenn ihr Programm unterscheiden könnte, ob es von Workbench oder CLI gestartet wurde. Dafür wird in "*wbstartup.h*" folgende Variable deklariert:

```
unsigned short _wbflag;
```

`"wbmain"` setzt diese Variable auf 1, so daß Ihr Programm einfach mit

```
if (_wbflag)
  von Workbench
else
  vom CLI
```

prüfen kann, wie es gestartet wurde.

So weit, so gut - aber wenn Ihr Programm die Standard-Ein-/Ausgabe benutzt (sei es nun mit `"cout"/"cin"` oder `"printf"/"scanf"`), fehlt immer noch ein Fenster dafür. Falls Sie ein solches wünschen, setzen Sie einfach VOR das `"include <wbstartup.h>"` eine Deklaration wie die folgende:

```
#define WBWINNAME "con:0/10/640/120/Demo"
```

Das Gerät "CON:" entspricht einem gewöhnlichen Textwindow, wie es auch das CLI bzw. eine Shell benutzt. Die dahinterstehenden Argumente geben (in dieser Reihenfolge) die X- und Y-Koordinate, die Breite, Höhe sowie den Titel des Windows an. Mit der obigen Deklaration wird also auf dem Workbench-Screen am linken Bildschirmrand und mit 10 Punkten Abstand vom oberen Rand ein Fenster namens "Demo" geöffnet, das 640 Punkte breit und 120 Punkte hoch ist. Beachten Sie bitte, daß unser `"wbmain"` aussteigt, wenn dieses Window nicht geöffnet werden kann, und daß nur eine Bildschirmgröße von 640 mal 200 Punkten (das entspricht dem HIRES-Noninterlaced-Modus auf NTSC-Rechnern ohne jeden Overscan) garantiert wird.

Die `"wbmain"`-Funktion öffnet also, wenn eine solche Deklaration vorliegt, eine entsprechende Datei und leitet Ein- und Ausgabe darauf um:

```
#ifdef WBWINNAME
// Ausgabe in Window umleiten:
if(!freopen(WBWINNAME, "w", stdout))
```

```
  return;
  // Eingabe aus selbem Window:
stdin->Filehandle = stdout->Filehandle;  // THIS ASSIGNMENT WAS
  // MADE BY TRAINED EXPERTS. DO NOT TRY THIS AT HOME!
#endif
```

Das Umleiten von **"stdout"** mit der ANSI-C-Funktion **"freopen"** ist standardmäßig und sicher. Anschließend wird aber ein ziemlich verwegener MaxonC++-Trick angewendet, um **"stdin"** dieselbe Datei zuzuweisen, indem mit Hilfe der Definitionen aus **"<streamdefs.h>"** die DOS-Filehandle manipuliert wird. Dies sollten Sie nur machen, wenn sie mit **"streamdefs.h"** vertraut sind und wissen, was sie tun.

Zu guter letzt folgt hier ein voll shell- und workbenchtaugliches Programm, das seine Argumente ausgibt. Beim WB-Start gibt es dem Benutzer (sofern man bei einem solchen Prograämmchen überhaupt von "benutzen" reden kann) Gelegenheit, die Ausgabe in Ruhe zu lesen, indem es am Ende wartet, bis <Return> gedrückt wird, denn das Ausgabefenster wird am Programmende automatisch sofort geschlossen.

```
#define WBWINNAME "con:0/10/640/120/Demo"
#include <wbstartup.h>
#include <stdio.h>

void main(int argc, char *argv[])
{
  for (int i = 0; i<argc; ++i)        // entweder Shell-Parameter
  printf("%2d : %s\n", i, argv[i]); // oder selektierte Icons

  if (_wbflag)    // diese Variable enthält ein Flag, das angibt,
  {               // ob das Proggi von der Workbench gestartet wurde
    printf("[CR] drücken: ");
    getchar();
  }
  exit(0);
}
```

7. Templates

7.1 Einführung

Templates sind die wohl wichtigste Neuerung des 3.0-Standards. Dabei handelt es sich um Schablonen, aus denen der Compiler selbstständig Klassen und Funktionen erzeugen kann.

In Abschnitt 3.4 wurde gezeigt, wie man aus einem allgemeinen Listentyp durch Ableitung Listen spezieller Elementtypen erzeugen kann. Dieser Weg ist zwar gangbar, aber doch ziemlich umständlich - man denke dabei an die virtuellen Funktionen, die man beim Ableiten immer wieder von Hand definieren muß. Viel einfacher wäre es doch, wenn man diesen Vorgang so weit automatisieren könnte, daß der Compiler letzten Endes selbst aus Beschreibungen wie `"Liste<int>"` oder `"Liste<Eintrag>"` entsprchende neue Klassen generieren kann - und genau das geht mit Klassen-Templates.

Die Bibliotheksfunktion `"qsort"` sortiert Vektoren beliebiger Elemente. Ihre Benutzung ist aber relativ umständlich, da man immer zuerst eine Vergleichsfunktion definieren und zu allem überfluß auch noch die Größe der Elemente angeben muß. Es wäre erheblich einfacher, wenn man eine Sortierfunktion mit mehr oder weniger beliebigen Argumenten aufrufen könnte und der Compiler dann selbsttätig die benötigten Funktionen erzeugen würde. Auch so etwas ist möglich, und zwar mit Funktions-Templates.

Zuerst befassen wir uns aber mit Klassen-Templates:

7.2 Klassen-Templates

7.2.1 Ein einfaches Beispiel

Klassen-Templates sind immer dann besonders nützlich, wenn man sogenannte Container-Typen definieren möchte. Das sind Klassen wie z. B. Listen, Arrays, Suchbäume oder Mengen, die Daten anderer Typen als Elemente enthalten und die man gerne so allgemein definieren möchte, daß diese Elemente nahezu beliebige Typen haben können.

Als einfaches Beispiel soll hier zuerst ein Vektor-Template implementiert werden. Wie Sie bisher sicher schon gemerkt haben, ist das von C ererbte Vektorkonzept von C++ nicht immer der Weisheit letzter Schluß. Die folgende, ganz konventionell programmierte Klasse implementiert einen `"int"`-Vektor mit 20 Elementen und der kleinen Besonderheit, daß der Index-Wertebereich von 1 bis 20 geht und zur Laufzeit überprüft wird:

```
class IntVec20
{
  int vec[20];
public:
  int &operator[](int i)
    { if (i >= 1 && i <= 20)
```

```
        return vec[i-1];
      else
        exit(-1);
    }
};

IntVec20 v1;
```

So weit, so gut. Das Problem bei der Sache ist aber, daß wir alles nahezu unverändert noch mal eintippen müssen, wenn wir einen Vektor von 50 "double"-Werten haben. Solcherlei Mühe ersparen wir uns fortan, indem wir hier einfach ein Template definieren:

```
template<class T, int anz> class Vector
{
  T vec[anz];
 public:
  T &operator[](int i)
    { if (i >= 1 && i <= anz)
        return vec[i-1];
      else
        exit(-1);
    }
};

Vector<int,20> v1;
Vector<double,50> v2;
```

Offensichtlich beginnt eine Template-Deklaration also mit dem neuen Schlüsselwort **"template"**. Dahinter folgt, in spitzen Klammern, die in C++ bisher noch nicht aufgetreten sind, eine Art Parameterliste. In einer solchen Liste kann es zwei Arten von Parametern geben:

- Typ-Parameter (**"class T"**) werden stets durch das Schlüsselwort **"class"**, gefolgt von einem einfachen Bezeichner, dargestellt. Wird das Template später benutzt, ist an dieser Stelle eine Typbeschreibung anzugeben. Man beachte, daß es sich bei dem Argument später keineswegs zwingend um eine Klasse handeln muß - im obigen Beispiel werden z. B. **"int"** und **"double"** übergeben. Das Schlüsselwort **"class"** wird hier also lediglich benutzt um anzuzeigen, daß es sich um einen Typparameter handelt.
- Wert-Parameter (**"int anz"**) sehen wie Parameter einer Funktion aus. Wird das Template benutzt, ist hier ein konstanter Ausdruck einzusetzen.

Hinter der Parameterliste folgt eine ganz normale Klassendefinition, bei der man einfach so tut, als seien die Parameter als Typ bzw. Konstanten definiert. Anstelle einer **"class"** können Sie hier auch eine **"struct"** oder **"union"** definieren.

Nachdem Sie auf diese Weise ein Template namens `"Vector"` definiert haben, können Sie es benutzen: Der Template-Name, gefolgt von einer Argumentliste, kann wie ein ganz normaler Typ- oder Klassenname benutzt werden, z. B. auch so:

```
Vector<char*,100> *vptr;

typedef Vector<int,20> IntVec20;

Vector<Vector<double,10>,10> dvecvec;
```

Im letzten Beispiel wurde das Template sogar als sein eigenes Argument benutzt, um einen Vektor von Vektoren zu erhalten. Templates stehen also absolut gleichberechtigt neben "normalen" Datentypen.

7.2.2 Deklarationen und Definitionen

Es ist einmal wieder an der Zeit, auf einigen Formalismen herumzureiten. Zunächst sollte erwähnt werden, daß Templates grundsätzlich nur auf Dateiebene und somit nicht lokal zu Klassen oder Funktionen definiert werden dürfen:

```
struct S {
  template <class T> class Valsch ... // ERROR!
```

Die Template-Namen liegen im globalen Namens-Scope, zusammen mit Namen von Funktionen und Variablen. Es ist auch nicht erlaubt, einen Templatenamen wie einen Funktionsnamen zu überladen:

```
template <class t>          class Tpl { };
template <class u,class v> class Tpl { };  // ERROR!
```

Bekanntlich ist es möglich (und wird auch oft praktiziert), eine Klasse zuerst nur zu deklarieren (z. B. `"class C;"`) und erst später wirklich zu definieren (`"class C { ...};"`). In der Zwischenzeit darf man z. B. Zeiger oder Referenzen auf `"C"` benutzen, aber z. B. kein Objekt dieser Klasse definieren. Ganz analog verhält es sich mit Definiton und Deklaration von Klassen-Templates:

```
template <class T> class C;     // Template-Deklaration

C<int> *cip;              // OK
typedef C<int> cit;       // OK
C<int> ci;                // ERROR: Data type undefined

template <class T> class C      // Template-Definition
{ public:
   T t1, t2; };

C<int> cj;                // OK
```

Zwischendurch ein Hinweis zur Nomenklatur: Eine Template-Definition wie

```
template <class T, int j> class C ...
```

definiert ein Klassen-Template `"C"`. Wenn wir später das Template benutzen, um eine neue Klasse zu erhalten, z. B.

```
C<int,42>,
```

dann nennt man das Ergebnis eine Template-Klasse.

Template-Klassen sind identisch, wenn ihre Argumente übereinstimmen. Beispiel:

```
template <class T, int j> class C;

typedef C<int,   42>   ctype;
typedef C<signed,2*21> ctype;  // OK, Typen identisch
typedef C<long,  41+1> ctype;  // ERROR, anderer Typ
```

Konstruktoren und Destruktoren von Template-Klassen heißen wie die Templates:

```
template <class T> class C
{  T t1;
 public:
   C(T);
   ~C();
};
```

Während der Definition eines Klassen-Templates ist der Templatename (im folgenden Beispiel wieder einmal `"C"`) identisch mit der jeweiligen Template-Klasse, kann aber trotzdem noch zur Erzeugung anderer Template-Klassen benutzt werden:

```
template <class T> class C
{
 public:
   C      *next;      // Zeiger auf selbe Klasse
   C<T>   *tptr;      // so geht's auch
   C<int> *iptr;      // ...aber so etwas geht auch.
};
```

Natürlich können Klassen-Templates auch abgeleitet werden, gleichermaßen von irgendwelchen konstanten Klassen oder ihren Argumenten:

```
class Base1
{
  // ...
};

template<class Base2> class Derived
  : public Base1, public Base2
{
  // ...
};
```

Zu guter letzt sei noch darauf hingewiesen, daß von MaxonC++ Template-Definitionen zunächst nur sehr flüchtig Syntax-gecheckt werden. Erst wenn das Template benutzt, d. h. eine Template-Klasse

erzeugt wird, wird das Template mit den eingesetzten Argumenten compiliert. Also hat der Compiler zunächst keinerlei Probleme mit Definitionen wie der folgenden:

```
template <int i> class Murx
  { Alles voll daneben, ey! };
```

Erst wenn später ein Typ wie `"Murx<17>"` benutzt wird, werden die zahlreichen Fehler gemeldet. Deshalb sollten Sie sich auch nicht wundern, wenn in einem Programm, das Templates enthält, die Fehler nicht in ihrer "natürlichen" Reihenfolge (strikt nach Zeilennummern geordnet) ausgegeben werden, sondern gerade so, wie der Compiler sie findet.

7.2.3 Member-Funktionen

Wie bei jeder Klasse können Sie auch bei Klassen-Templates Member-Funktionen direkt in der Klassendefinition definieren - aber dann gelten Sie natürlich automatisch als `"inline"`, was ja nicht immer wünschenswert ist. Will man eine Funktion außerhalb des Templates definieren, wählt man die bisher von Memberfunktionen bekannte Schreibweise - nur daß bei den qualifizierten Bezeichnern als Klassenname hier der Templatename samt Parameterliste zu wählen ist:

```
template <class T> class C1
{ public:
    void f(int);
    T g(const T&);
    C1(T);
};

template <class T> void C1<T>::f(int i)
{
  // ...
}

template <class typ> typ C1<typ>::g(const typ&)
{
  // ...
}

template <class T> C1<T>::C1(T t)
{
  // ...
}
```

Wie Sie anhand der Definition von `"g"` erkennen können, müssen die Argumentnamen nicht zwangsläufig mit der ursprünglichen Definition übereinstimmen.

Es ist allerdings auch erlaubt, Funktionen für konkrete Klassen speziell zu definieren:

```
#include <iostream.h>

template <class T> struct S
{ T t1;
```

```
  void f(); };

template <class T> void S<T>::f()
{ cout << "Class: " << sizeof(T) << "\n"; }

void S<char>::f()
{ cout << "Spezial-Funktion für CHAR\n"; }

S<int> s1;
S<char> s2;
S<double> s3;

void main()
{
  s1.f();  // normale Template-Funktion
  s2.f();  // spezielle Funktion
  s3.f();  // normale Template-Funktion
}
```

Für die Klasse `"S<char>"` wurde hier eine Funktion `"f"` definiert, was Vorrang vor der allgemeinen Funktionsdefinition des Templates hat, welche der Compiler für alle anderen Template-Klassen verwendet.

7.2.4 Statische Member

Sie erinnern sich: ein statischer Member einer Klasse wird (naheliegenderweise) mit der Speicherklasse `"static"` deklariert und existiert dann genau einmal für alle Objekte der Klasse zusammen.

Damit dieser Member aber auch noch tatsächlich irgendwo real existiert und vom Linker gefunden werden kann, muß er in genau einer übersetzungseinheit des Programms auch noch definiert werden.

Auch Template-Klassen dürfen statische Member besitzen. Beispiel:

```
template <class T> class C
{       T t2;     // gewöhnlicher Member
  static T t1;    // statischer Member
};

C<int> ci;
C<char> cj;

int C<int>::t1 = 42;
```

Für jede einzelne Templateklasse (und nicht etwa gemeinsam für alle Klassen eines Templates) wird ein statischer Member erzeugt, also eine char-Variable für `"C<char>"` und eine int-Variable für `"C<int>"`. Der kleine Unterschied zu statischen Membern in "gewöhnlichen" Klassen ist, daß die Member nicht explizit definiert werden müssen. Im obigen Programm existiert die char-Variable `"C<char>::t1"` also tatsächlich. Auf Wunsch kann der Programmierer diese Member aber auch

explizit definieren, wie es oben bei `"C<int>::t1"` geschehen ist. Dabei kann man dann optional auch in gewohnter Weise eine Initialisierung vornehmen.

Falls es für einen statischen Member einer Templateklasse aber keinen Default-Initialisierungswert bzw. -Konstruktor gibt, wird der Member nicht automatisch erzeugt und muß folglich irgendwo explizit definiert werden. Stellen Sie sich im obigen Beispiel z. B. eine Templateklasse `"C<int&>"` vor. Der statische Member `"t1"` hätte dann folgerichtig den Referenztyp `"int&"`, für den immer eine Initialisierung erforderlich ist. Die Variable `"int &C<int&>::t1"` ist folglich explizit zu definieren.

7.2.5 Friends

Bei allem, was mit Friend-Deklarationen oder sonstwie mit Zugriffsrechten zu tun hat, sollten Sie sich immer vor Augen halten, daß Templates nicht viel anderes sind als Makros, in denen Argumente textuell ersetzt werden.

Die Friend-Funktionen einer Templateklasse sind nicht automatisch Templatefunktionen (die werden in diesem Handbuch sowieso erst später behandelt). Beispiel:

```
template <class T> class C
{
  friend char *f1(char *str);
  friend void  f2(const C<T> &c);
};
```

Hier gibt es genau eine Funktion namens `"f1"`, die Friend sämtlicher Templateklassen des Templates `"C"` ist. Außerdem gibt zu jeder einzelnen Templateklasse eine eigene Funktion namens `"f2"`, die als Argument eine Referenz auf eben jene Klasse besitzt. Für `"f2"` wird man dann sinnvollerweise ein Funktionstemplate einführen, denn andernfalls müßte man ja alle diese Funktionen von Hand definieren.

Wenn eine gewöhnliche Klasse Friend einer Templateklasse wird, gibt es nichts besonders Aufregendes zu beachten. Wenn jedoch eine andersrum eine Templateklasse Friend werden soll, ist wieder einmal der Unterschied zwischen Templateklasse und Klassentemplate zu beachten:

```
template <class T> class Freund
{
  // Inhalt interessiert nicht
};

class Normal
{
  friend class Freund<char*>;   // OK
  friend class Freund;          // ERROR
};

template <class S> class Temp
{
  friend class Freund<S>;       // OK
```

```
  friend class Freund;          // ERROR
};
```

In beiden Fällen ist es nicht möglich, einem ganzen Klassentemplate kollektiv die Freundschaft anzubieten, sondern nur einer ganz bestimmten, einzelnen Templateklasse.

7.3 Funktions-Templates

7.3.1 Beispiel: Sortieren von Vektoren

Bei den bisher beschriebenen Klassentemplates werden auch Memberfunktionen über Templates generiert. Es gibt allerdings auch die Möglichkeit. globale Funktionen (also solche auf Dateiebene) über sogenannte Funktionstemplates erzeugen zu lassen.

Als Beispiel für ein Funktionstemplate sehen wir uns die folgende (zugegebenermaßen nicht besonders effektive) Sortierfunktion an:

```
template <class T> void bubblesort(T *vec, int size)
{
  for (int i = size-1; i>0; --i)
    for (int j = 0; j < i; ++j)
      if (vec[j] > vec[j+1])
        { // benachbarte Elemente vertauschen:
          T hilf    = vec[j];
          vec[j]    = vec[j+1];
          vec[j+1] = hilf;
        }
}
```

Auf diese Weise können wir nahezu beliebige Vektoren sortieren lassen - vorausgesetzt, der Vergleichsoperator `">"` ist auf den Vektorelementen sinnvoll definiert:

```
#include <stream.h>

// Beispieldaten:
int iv[] = { 17, 4, 21, 26, 7, 31, 47, 11, 0, 8, 15 };
char cv[] = { 'c', 'a', 'd', 'i', 'l', 'l', 'a', 'c' };
char *sv[] = { "Steck", "deinen", "Kopf", "in", "ein", "Schwein" };

void main()
{
  int i;

  bubblesort(iv, 11);
  bubblesort(cv, 8);
  bubblesort(sv, 6);

  for (i=0; i<11; ++i)
    cout << iv[i] << " ";
  cout << "\n\n";
```

```
  for (i=0; i<8; ++i)
    cout << cv[i] << " ";
  cout << "\n\n";

  for (i=0; i<6; ++i)
    cout << sv[i] << " ";
  cout << "\n\n";
}
```

7.3.2 Deklaration und Definition

Jedes Funktionstemplate kann, genau wie ein Klassentemplate, beliebig oft deklariert werden, also z. B. in dieser Art:

```
template <class T> void bubblesort(T *vec, int size);
```

Damit wird dem Compiler die Existenz eines solchen Templates bekannt gemacht. Außerdem muß es im Programm genau eine Definition (also einen Anweisungsteil für die Funktion) zu jedem deklarierten Template geben.

Bei den Template-Argumenten eines Funktionstemplates muß es sich um Typargumente handeln. Die folgende Deklaration ist also fehlerhaft:

```
template <class T, int size> void nocheinsort(T vec[size]);
```

Bei Klassentemplates werden die Argumentwerte bekanntlich explizit angegeben, während der Compiler die Argumenttypen eines Funktionstemplates aus den Argumenten des Funktionsaufrufs herausklauben muß. Deshalb muß jedes Templateargument auch in der Parameterliste auftauchen:

```
template <class A, class B, class C> A fun(B b)  // ERROR
  { C c1 = b;
    return 2*c1
  }
```

Aus einem Aufruf wie `"fun(42)"` kann der Compiler hier nicht erkennen, mit welche Typen die Argumente `"A"` und `"C"` belegt werden sollen. Deshalb ist dieses Funktionstemplate unzulässig.

7.3.3 Handgestrickt kontra compilergeneriert

Templates haben kein Exklusivrecht an ihrem Namen. Der Programmierer kann einem Funktionstemplate beliebig viele Funktionen des gleichen Namens hinzufügen.

Als Beispiel nehmen wir das Bubblesort-Template aus 7.3.1 noch einmal unter die Lupe. Dieses Template ist nur sinnvoll, wenn der Operator `">"` auf dem Argumenttyp passend definiert ist. Auf Klassen ist dies z. B. nicht der Fall (man kann aber `">"` entsprechend überladen) , und bei `"char*"` vergleicht `">"` die Speicheradressen der Zeichenketten, was meist nicht das Gewünschte ist. Wir verbessern das Programm deshalb folgendermaßen:

```
template <class T> inline int greater(const T &t1, const T &t2)
  { return t1 > t2; }

template <class T> void bubblesort(T *vec, int size)
{
  for (int i = size-1; i>0; --i)
    for (int j = 0; j < i; ++j)
      if (greater(vec[j], vec[j+1]))
        { // benachbarte Elemente vertauschen:
          T hilf    = vec[j];
          vec[j]    = vec[j+1];
          vec[j+1] = hilf;
        }
}
```

Der Vergleichoperator `">"` ist hier also durch eine ominöse Funktion `"greater"` ersetzt worden. Im allgemeinen Fall wird `"greater"` durch ein Template erzeugt. Wnn man aber auf bestimmten Typen spezielle Vergleiche realisieren möchte, kann man eine maßgeschneiderte Vergleichsfunktion hinzudefinieren:

```
#include <stream.h>
#include <string.h>

inline int greater(char *s1, char *s2)
  // spezielle Vergleichsfunktion für Zeichenketten:
  { return stricmp(s1,s2) > 0; }

// Beispieldaten:
int iv[] = { 17, 4, 21, 26, 7, 31, 47, 11, 0, 8, 15 };
char *sv[] = { "Steck", "deinen", "Kopf", "in", "ein", "Schwein" };

void main()
{
  int i;

  bubblesort(iv, 11);
  bubblesort(sv, 6);

  for (i=0; i<11; ++i)
    cout << iv[i] << " ";
  cout << "\n\n";

  for (i=0; i<6; ++i)
    cout << sv[i] << " ";
  cout << "\n\n";
}
```

Hier wird für Zeichenketten eine sinnvoller, alphabetische Vergleichsoperation eingeführt, während für den `"int"`-Vektor der aus dem Template `"greater"` erzeugte Standardvergleich benutzt wird.

Damit wären wir bei der Frage gelandet, wann in C++ eine Templatefunktion generiert und wann eine halbwegs passende "echte" Funktion benutzt wird, und somit wird es höchste Zeit für das folgende Kapitel:

7.3.4 Argumemt-Matching beim Aufruf von Templatefunktionen

7.3.4.1 Benutzung von Funktionstemplates

In Abschnitt 4.1.2.2 wurde das genaue Verfahren beschrieben, nach dem C++ beim Aufruf eines überladenen Funktionsnamens feststellt, welche Funktion aufgerufen werden soll. Dieses Verfahren war zugegebenermaßen reichlich kompliziert. Nun stehen wir aber vor einer noch etwas unübersichtlicheren Situaton: Wir haben es unter Umständen nicht nur mit einem (beliebig überladenen) Funktionsnamen zu tun, sondern können mit Hilfe eines Templates noch unendlich viele Funktionen des gleichen Namens hinzufügen. Aber keine Angst: theoretisch wird es nicht viel schwieriger als bisher, und praktisch gesehen werden Sie beim Programmieren recht wenig beachten müssen.

Zunächst betrachetn wir einmal den etwas einfacheren und nichtsdestotrotz wichtigen Fall, daß wir nur ein Template und keine explizit deklarierten Funktionen vor uns haben. Unter welchen Bedingungen kann eigentlich eine Templatefunktion zu einem Funktionsaufruf generiert werden?

Das ist, kurz gesagt, nur möglich, wenn die Typen der Argumente exakt mit denen der Parameterliste übereinstimmen bzw. durch geeignete Belegung der Templateargumente übereinstimmend gemacht werden können. Dabei sind allenfalls triviale Konvertierungen (siehe 4.1.2.1) wie `"T[] -> T*"` oder `"T -> const T"` zulässig:

```
template <class T> void f1 (T t1, T t2);

void test()
{
  f1(17, 4);      // OK
  f1(17, 4.0);    // Error: Arguments do not match function template
}
```

In diesem Beispiel ist der erste Funktionsaufruf offensichtlich richtig, während beim zweiten nicht klar ist, ob das Templateargument `"T"` nun `"int"` oder `"double"` sein soll. Theoretisch könnte der Compiler auf Nummer sicher gehen und sich für `"double"` entscheiden. Solche überlegungen sind vom C++-Sprachstandard aber nicht vorgesehen. Es wird exakte übereinstimmung der Argumenttypen mit den Parametern des Funktionstemplates erwartet, und `"int"` ist mit `"double"` nun einmal nicht annähernd identisch.

Durch diese Regel kann es zu unerwarteten Effekten kommen:

```
template <class T> void f2 (T *vect, unsigned size);

void test()
{
  char v[] = { 'h', 'a', 'l', 'l', 'o' };

  f2(v, 5);      // ERROR
  f2(v, 5U);     // OK
}
```

Beim ersten Versuch stimmt der Typ (`"int"`) des zweiten Arguments (also der `"5"`) nicht exakt mit dem Typ des Parameters `"size"` überein. Das Dogma von der exakten Typgleichheit gilt also auch für solche Parameter, deren Typen sozusagen `"konstant"` sind und nichts mit irgendwelchen Templateargumenten zu schaffen haben.

7.3.4.2 Mischung von Funktiontemplates und "echten" Funktionen

Im Beispielprogramm aus 7.3.3 wurden bereits Funktionstemplates mit explizit deklarierten Funktionen gleichen Namens gemischt. Wenn eine solche Funktion aufgerufen wird, wird nach dem folgenden Verfahren die passende Funktion bzw. Templatefunktion ermittelt:

1. Es wird nach einer (explizit deklarierten) Funktion gesucht, deren Parameter exakt mit den aktuellen Argumenten übereinstimmen. Triviale Konvertierungen sind dabei großzügigerweise erlaubt.
2. Wird keine geeignete Funktion gefunden, so wird versucht, eine Templatefunktion zu erzeugen. Auch hierbei ist naturgemäß exakte Typgleichheit Pflicht.
3. Wenn die beiden ersten Versuche fehlgeschlagen sind, wird mit den Funktionen (also den explizit deklarierten) ein ganz normales Argument-Matching wie in 4.1.2.2 durchgeführt.
4. Hilft das alles nichts, ist der Funktionsaufruf ein Fehler.

In den Schritten (1) und (3) kann sich herausstellen, daß der Funktionsaufruf mehrdeutig ist. Die Generierung einer Templatefunktion ist hingegen immer eine eindeutigt Angelegenheit.

Apropos eindeutig: Sie können die Verpflichtung zur Exaktheit umgehen, indem Sie hemmungslos Funktionen hinzudefinieren:

```
template <class T> void f1 (T t1, T t2);

void f1(double, double);

void test()
{
  f1(17, 4);      // OK, f1(int,int)
  f1(17, 4.0);    // Jetzt auch O.K., f1(double,double)
}
```

Der zweite Funktionsaufruf ist jetzt korrekt, weil nach Schritt (3) des Auflösungsalgorithmus ganz konventionell mit `"f1(double,double)"` gematcht werden kann. übrigens verpflichtet diese Deklaration Sie keineswegs, die deklarierte Funktion auch selbst zu definieren: Da die Funktionsdeklaration zum Template paßt, greift hier der Compiler bei Bedarf ein.

7.3.5 Generierung von Templatefunktionen

Bei den bisher betrachteten kleinen Beispielen war es für den Compiler (und erst recht für den Programmierer) noch eindeutig, welche Templatefunktionen generiert werden müssen. Wenn das Programm aber aus mehreren übersetzungseinheiten (Modulen) besteht, ist dies nicht mehr so einfach.

MaxonC++ kann dabei prinzipiell zwei verschiedene Strategien verfolgen und überläßt Ihnen die Auswahl. Beachten Sie dazu bitte auch Abschnitt 2.5.2.6.10 im MaxonC++ Anwenderhandbuch.

Letzten Endes geht es dabei lediglich um Funktionstemplates wie

```
template <class T> T f1(T t1)
  { return t1+1; }

template <class T> void f2(T t1)
  { Jawollja! }
```

Aus dem ersten Template kann nur dann eine Templatefunktion erzeugt werden, wenn das Argument numerisch oder ein Pointer ist (oder meinetwegen auch eine Klasse, auf der `"+"` überladen ist). Aus dem Template `"f2"` kann offensichtlich niemals eine korrekte Funktion generiert werden.

MaxonC++ erzeugt in jeder übersetzungseinheit alle Templatefunktionen, die dort benutzt oder deklariert, aber nicht schon vom Programmierer definiert wurden. Dabei können natürlich unnötige Funktionen generiert werden, zu denen es in einem anderen Modul schon eine explizite Definition gibt. Das ist an und für sich nicht so schlimm, da der Linker solchen Ballast später wieder herauswirft. Problematisch wird es aber, wenn in einer Templatefunktion ein Fehler auftritt.

Es liegt nun an Ihnen, wie der Compiler in einer solchen Situation verfahren soll. Zum einen kann er dann die Fehlermeldung unterdrücken und die Erzeugung der fraglichen Funktion schlicht unterlassen. Bei einem korrekten Programm erhalten Sie so ein auf jeden Fall korrektes und standardgemäßes Ergebnis. Wenn sich aber ein Fehler in eine Ihrer Templatedefinitionen eingeschlichen hat, erfahren Sie davon erst, wenn der Linker die entsprechende Funktion nicht findet, und der kann Ihnen dann auch nicht mehr sagen, wo genau der Fehler lag.

Deshalb gibt es für die Programmentwicklung eine zweite Strategie, die in den meisten praktischen Fällen auch zu korrekten Ergebnissen führen wird. Auch hier wird versucht, alle theoretisch erforderlichen Templatefunktionen zu generieren. Die dabei auftretenden Fehler werden aber in gewohnter Weise gemeldet. Für eine weitere Diskussion dieser Wahlmöglichkeit verweise ich Sie erneut auf Abschnitt 2.5.2.6.10 des Benutzerhandbuchs.

7.4 Beispielprogramm: Schon wieder verkettete Listen

In Abschnitt 2.7.6 wurde eine simple Implementierung einer doppelt verketteten Liste vorgestellt, und in Kapitel 3.4 haben wir die Angelegenheit über Vererbung etwas eleganter gelöst - Ich hoffe, daß Sie diese Programme nicht schon völlig vergessen haben. Einer der Hauptvorteile der Alzheimer-Krankheit ist ja, daß man jeden Tag neue C++-Features kennenlernt (allerdings bin ich mir nicht sicher, ob überhaupt ein gesunder Mensch jemals C++ in seiner Gesamtheit erfassen wird). Es drängt sich geradezu auf, das gleiche Problem über Templates zu lösen - und Sie werden sehen, daß es damit noch einfacher wird. Damit Kapitel 4 nicht völlig für die Katz war, benutzen wir auch die kleine Stringbibliothek aus Abschnitt 4.2.3. Dadurch können die Namen und Telefonnummern beliebig lang werden.

Damit es nicht allzu langweilig wird, fangen wir noch mal ganz grundsätzlich von vorn an. Die Elemente einer typischen sortierten Liste haben meist einen `"Schlüssel"`, nach dem sortiert bzw. gesucht wird, und die mit dem Schlüssel assoziierten eigentlichen Daten. Oder, um noch einmal auf unserem Telefonbuch-Beispiel herumzureiten: Die Liste wird sortiert und durchsucht nach Namen, und zu jedem Namen gehört dann eine Telefonnummer. Vereinfacht sieht das Template für einen Listeneintrag dann so aus:

```
template <class K, class D> class Listenelement
{
   K key;    // Schlüssel
   D data;   // assoziierte Daten
};
```

Wir gehen davon aus, daß die Daten des `"key"`-Typs mit den Operatoren `"<"` und `"=="` vergleichbar sind und über `"cout"` ausgegeben werden können - falls dem nicht so ist, soll der Benutzer unserer Templates die Operatoren eben entsprechend überladen. Damit steht auch schon das Skelett unseres Programms. Den ganzen Rest entnehmen sie bitte dem nachfolgenden kommentieten Listing.

```
//  Ein Listen-Template
//
//  Jens Gelhar 16.09.94

#include <stream.h>
#include <tools/str.h>

// * * * generische Datentypen * * *

template <class K, class D> class Liste;

template <class K, class D> class Listenelement
{ private:
    Listenelement *next, *prev;    // Vorgänger und Nachfolger
    K key;          // Schlüssel, nach dem gesucht und sortiert wird
    D data;         // der eigentliche Inhalt

    virtual ~Listenelement() { }
```

```
    // virtueller Destruktor - hier nur ein Dummy, kann aber
    // in abgeleiteten Klassen überdeckt werden

  public:
    // kleine Extras: andere Klassen dürfen "next" und
    // "prev" auslesen
    Listenelement<K,D> *getnext() const
      { return next; }
    Listenelement<K,D> *getprev() const
      { return prev; }

  friend class Liste<K,D>;  // Diese später deklarierte Klasse
                            // braucht Zugriff auf private Member
};


template <class K, class D> class Liste
{ private:
    Listenelement<K,D> *anfang, *ende;  // erstes und letztes Element
                                        // der Liste
  public:
    Liste();           // Konstruktor zur Initialisierung
    ~Liste();          // Destruktor (löscht alle Listenelemente)

    void ausgabe();    // gibt die ganze Liste aus

    void einfuegen(const K &k, const D &d);
    // fügt ein neues Element in die Liste ein

    Listenelement<K,D> *suchen(const K &k);
    // sucht ein Element mit dem Schlüssel k und liefert Zeiger
    // darauf bzw. Null, wenn kein solches Element existiert.

    void loeschen(Listenelement<K,D> *);
    // hängt ein Element aus der Liste aus und löscht es
    // mit "delete"
};

template<class K, class D> Liste<K,D>::Liste()
{
   anfang = 0;
   ende = 0;
}

template<class K, class D> Liste<K,D>::~Liste()
{
   Listenelement<K,D> *p = anfang;

   while(p)
     { Listenelement<K,D> *hilf = p;
       p=p->next;
       delete hilf;
```

```
    }
}

template<class K, class D> void Liste<K,D>::ausgabe()
{
   for(Listenelement<K,D> *l = anfang; l; l = l->next)
      cout << l->key << " " << l->data << "\n";
}

template<class K, class D>
   Listenelement<K,D> *Liste<K,D>::suchen(const K &k)
{
   for(Listenelement<K,D> *e = anfang;
       e && e->key != k;
       e = e->next);
   return e;
}

template<class K, class D>
  void Liste<K,D>::einfuegen(const K &k, const D &d)
{
   // Position suchen:
   Listenelement<K,D> *neu, *pos = anfang;
   while (pos && pos->key > k)
      pos = pos->next;

   // neues Element erzeugen:
   neu = new Listenelement<K,D>;
   if (!neu) return;
   neu->key = k;
   neu->data= d;

   // Element an Position "pos" in Liste einfügen
   // (prinzipiell genau wie in der ersten Programmversion)

   if (pos == 0)
    // Sonderfall: Anfügen an das Ende der Liste
    {
      Listenelement<K,D> *alt = ende;     // altes Listenende

      if (alt != 0)
      // allgemeiner Unter-Fall: wirklich anhängen
      {
        alt->next = neu;
        neu->prev = alt;
        neu->next = 0;
        ende = neu;
      }
      else
      // spezieller Spezialfall: Liste ist noch leer
      { neu->next = 0;
        neu->prev = 0;
```

```
        anfang = neu;
        ende = neu;
      }
    }
   else
    if (pos->prev == 0)
    // Sonderfall: Neues Element an Listenanfang hängen
    {
      pos->prev = neu;
      neu->next = pos;
      neu->prev = 0;
      anfang = neu;
    }
    else
    // allgemeiner Fall: Element in Liste einhängen
    {
      Listenelement<K,D> *vor = pos->prev;
      // "neu" zwischen "vor" und "pos" einhängen:
      vor->next = neu;
      neu->prev = vor;
      neu->next = pos;
      pos->prev = neu;
    }
}

template<class K, class D>
   void Liste<K,D>::loeschen(Listenelement<K,D> *E)
  // Löscht das Element, auf das "E" zeigt, aus der Liste
{
   // Vorgänger unf Nachfolger des zu löschenden Elements
   Listenelement<K,D> *vor = E->prev, *nach = E->next;

   if (vor)
   // Element hat Vorgänger: Dann dessen Nachfolger umsetzen
     vor->next = nach;
   else
   // kein Vorgänger: Dann Listenanfang aktualisieren
     anfang = nach;

   if (nach)
   // Element hat Nachfolger: Dann dessen Vorgänger verändern
     nach->prev = vor;
   else
   // kein Nachfolger, also Listenende anpassen
     ende = vor;

   delete(E);
}

// * * * spezielle Datentypen * * *

typedef Listenelement <String,String> Eintrag;
```

```
typedef Liste <String,String> Telefonbuch;

// * * *  Hauptprogram  * * *

int main()
{
   // Eine Liste wird deklariert und initialisiert:

   Telefonbuch t;

   // Ein paar Beispieldaten werden eingefügt:

   t.einfuegen("Harley-Davidson", "001-414/342-4680");
   t.einfuegen("Boris Becker", "0815/4711");
   t.einfuegen("James Bond", "007/26731");
   t.einfuegen("Maxon", "06196/481813");
   t.einfuegen("Zaphod Beeblebrox", "00042/08154242");
   t.einfuegen("Die Allwissende Billardkugel", "0231/7281453");
   t.einfuegen("Queensr\377che", "795069");
   t.einfuegen("Fishbone", "4676152");

   // Ein Datensatz wird gesucht und ggf. gelöscht:

   Eintrag *e = t.suchen("Maxon");
   if (e)
     t.loeschen(e);
   else
     cout << "Name nicht gefunden!\n";

   // restliche Liste ausgeben:
   t.ausgabe();

  // Löschen der Liste geschieht automatisch durch Destruktor-Aufruf
}
```

8. Ausnahmebehandlung

8.1 Worum gehts?

Wenn ein größeres Programm läuft, tut es das nicht immer so glatt wie es sollte. Bullshit happens, und laut Murphy wird zwangsläufig so einiges schiefgehen: Mal läuft der Speicher über, dann kann ein Window oder Screen nicht geöffnet werden; ein anderes mal ist eine Library nicht vorhanden oder eine Datei nicht verfügbar. Falls Sie mit dem RAM-Ausbau Ihres Amiga etwas knauserig waren, muß ich Ihnen sicher nicht erzählen, daß so manches Programm gnadenlos abstürzt, wenn der Speicher nicht reicht - und dieses Problem gibt es natürlich nicht nur in der Amiga-Welt.

Warum eigentlich wird die Fehlerbehandlung bei Programmen oft so nachlässig gehandhabt? Weil es - Hand aufs Herz! - ganz einfach nervt, bei jeder kritischen Operation auftretende Fehler sauber abzufangen und die Programmausführung dann sauber fortzusetzen. Wenn z. B. eine Datei geschrieben werden soll, prüfen die weitaus meisten Programmierer durchaus, ob sie korrekt geöffnet werden konnte. Die wenigsten testen aber bei jeder Schreiboperation, ob sie korrekt durchgeführt wurde - ganz einfach, weil das für den Programmierer jede Menge Arbeit bedeutet.

Erschwerend kommt ja noch hinzu, daß der Code, in dem ein Fehler auftritt, oft von jemand ganz anderem geschrieben wird als der Programmteil, in dem er abgefangen werden muß. Tritt z. B. in einer Klassenbibliothek ein Speicherüberlauf oder ein anderer Allerweltsfehler auf, weiß der Entwickler der Bibliothek nicht, was er damit anfangen soll: Das ganze Programm mit einer Fehlermeldung abbrechen? Die Operation abbrechen und einen Fehlercode zurückgeben, der im Zweifelsfall vom aufrufenden Programm ohnehin ignoriert wird? Den Fehler irgendwie dem Anwender des Programms melden und sich so gut wie möglich durchwurschteln? Solche Unklarheiten führen oft zu „unerwarteten" Effekten.

Um die Zuverlässigkeit der Software zu erhöhen, besitzen einige moderne Programmiersprachen das Konzept der Ausnahmebehandlung („Exception Handling"). Der Programmierer soll die Möglichkeit erhalten, sein Programm nach jedem denkbaren Laufzeitfehler so sinnvoll wie möglich fortsetzen zu lassen. Zugegebenermaßen kann man das theoretisch in jeder beliebigen Programmiersprache, aber durch Ausnahmebehandlung kann der Programmierer das so einfach implementieren, daß eine gewisse Aussicht besteht, daß er es auch tut. Deshalb ist Exception Handling z. B. Bestandteil der Sprache ADA (Militärs hatten ja immer schon ein gesteigertes Sicherheitsbedürfnis) und wurde kürzlich auch in C++ aufgenommen. Neben den Templates ist Ausnahmebehandlung die zweite große Neuerung des 3.0-Standards und wird mit an Sicherheit grenzender Wahrscheinlichkeit auch in den geplanten ANSI C++ Standard eingehen.

Damit hier keine Mißverständnisse entstehen: Das C++ Exception Handling hat absolut nichts mit Prozessor-Exceptions zu tun, auch wenn eine ähnliche Idee dahintersteht. Ausnahmebehandlung ist ein echtes High-Level-Feature einer Luxus-Programmiersprache und vollkommen unabhängig von der unterliegenden Hardware. So, und wenn ich Ihnen jetzt noch verspreche, daß das ganze ziem-

lich einfach zu benutzen ist, sind Sie hoffentlich motiviert, sich dieses Kapitel aufmerksam durchzulesen.

8.2 Vom Werfen und Auffangen

Nehmen wir einmal an, eine Funktion ihres Programms entdeckt irgendwo einen Laufzeitfehler, mit dem sie nichts anfangen kann. Dies ist der rechte Zeitpunkt, um eine Ausnahme zu „werfen", wofür wir eine `"throw"`-Anweisung verwenden. Dadurch wird die Funktion abgebrochen und die Programmausführung in einem passenden „Handler" in der aufrufenden Funktion fortgesetzt.

Eine Mitteilung der Art „Ey Boß, wir haben ein Problem, aber ich weiß leider nicht, welches" ist im allgemeinen wenig hilfreich. Beim Schmeißen von Exceptions wird deshalb ein Ausdruck beliebigen Typs übergeben. Eine Funktion mit einer `throw`-Anweisung könnte z. B. so aussehen:

```
char *alloc_irgendwas()
 {
   char *mem = new char[26731];
   if (!mem)                          // Fehler aufgetreten:
     throw "Das ging in die Hose";    // also Exception werfen

   ... irgendwelcher sonstiger Code ...

   return mem;
 }
```

Der Ausdruck, der bei dieser Exception übergeben wird, ist hier also ein mehr oder weniger informativer String. Ebenso einfach wäre es, eine Fehlernummer zu übergeben, z. B. `"throw 42"`. In der Praxis wird man nicht so einfache Ausdrücke, sondern Objekte irgendwelcher Klassen verwenden, aber für den Anfang beschränken wir uns hier auf simple Beispiele.

Viel interessanter ist im Moment nämlich die Frage, was das Programm tut, nachdem die Ausnahme geworfen wurde. Normalerweise sieht ein Programmablauf ja so aus, daß das Hauptprogramm Funktionen aufruft, welche wiederum Funktionen aufrufen, die selbst wieder Aufrufe enthalten von Funktionen, die weitere Funktionen... - ich nehme an, sie haben verstanden, was ich sagen will. Irgendwo auf tiefer Ebene dieser vielfach verschachtelten Funktionsaufrufe kann nun jederzeit etwas schiefgehen, und laut Murphy wird es auch schiefgehen. Ein Laufzeitfehler macht es im Zweifelsfall unmöglich, sämtliche Prozeduren sinnvoll zu Ende zu führen. Es muß also eine geeignete Anzahl von Prozeduraufrufen kalt abgebrochen und auf einer geeigneten Ebene der Fehler behandlet und das Programm fortgeführt werden.

Einen Ausnahme-Handler modelliert man durch einen `„try"`-Block, der überall im Programm als Anweisung benutzt werden kann. Er hat ungefähr folgende Form:

```
void sicher()
 {
   IrgendwelcherCode();

   try      // dieses Symbol leitet den Try-Block ein
    {       // der Anweisungsteil ist immer eine Verbundanweisung
```

```
        Anweisung1();
        Anweisung2();
        // usw...
    }
    catch (int i)     // Ein Handler für Ausnahmen des Typs "int"
    {
        cout << "Fehler #" << i << "!\n";
        BehandleFehler(i);
    }
    catch (char*)
    {
        // Code für Behandlung von String-Fehlern
    }
    catch (const class Test &t)
    {
      // noch so ein Handler
    }

    RestlicherCode();
}
```

In der Funktion `"sicher"` taucht also plötzlich eine Try-Anweisung auf. Sie enthält einige Anweisungen, für die die nachfolgend beschriebenen Handler gelten sollen. Die Handler werden unmittelbar an die Try-Anweisung angehängt und mit dem Symbol `"catch"` eingeleitet. Zunächst aber werden die Anweisungen in der Try-Anweisung ausgeführt, und wenn dabei alles klar geht, wird hinter dem Try-Block weitergemacht, hier also bei `"RestlicherCode()"`.

Interessant wird es, wenn innerhalb der Try-Anweisung eine Ausnahme geworfen wird. Normalerweise wird dies nicht direkt textuell innerhalb des Try-Blocks geschehen - im Beispiel schon allein deshalb nicht, weil dort überhaupt keine Throw-Anweisung steht - sondern in einer ganz anderen Funktion, die direkt oder indirekt in der Try-Anweisung aufgerufen wird.

Es könnte z. B. sein, daß die Funktion `"Anweisung1()"` eine Funktion `"xxx()"` und die wiederum die oben dargestellte Funktion `"alloc_irgendwas()"` aufruft, und darin steht ja bekanntlich eine Throw-Anweisung mit einem String-Argument, das den Typ `"char *"` besitzt. Nehmen wir an, daß der entsprechende Laufzeitfehler auch auftritt und die Ausnahme folgerichtig geworfen wird. Nun werden alle aufgerufenen Funktionen abgebrochen und vom Laufzeitstack entfernt, bis auf einen Try-Block mit einem passenden Handler gestoßen wird. In unserem Fall befinden wir uns ja immer noch in der Ausführung des Try-Blocks aus der Funktion `"sicher()"`, welcher einen Handler für den Typ `"char *"` enthält. Wenn wir zusätzlich einmal unterstellen, daß auf tieferer Ebene kein weiterer passender Handler deklariert wurde, wird jetzt der Code dieser Catch-Anweisung ausgeführt. Hier sollte das Programm in einen möglichst sinnvollen Zustand versetzt werden, denn anschließend wird das Programm hinter dem Try-Block (`"RestlicherCode()"`) fortgesetzt, als wenn nichts passiert wäre.

8.3 Von Ausnahmen und ihren Typen

Eine Throw-Anweisung kann einen Ausdruck beliebigen Typs werfen. Dann werden alle zu dem Zeitpunkt begonnenen und noch nicht beendeten Try-Blöcke nach einem passenden Handler abgesucht, und der erste gefundene wird ausgeführt. Gibt es überhaupt keine passende Catch-Anweisung, steigt das Programm gnadenlos aus.

Genau das habe ich Ihnen schon einmal mehr oder weniger genauso erzählt. Was aber ist ein „passender" Handler? Trivialerweise kann eine Exception eines Typs `"T"` von einem Handler gleichen Typs, also einer Anweisung wie `"catch(T t1)"`, aufgefangen werden. Aber auch einige Typumwandlungen sind möglich, allerdings leider nicht alle in C++ bekannten Umwandlungen. Die korrekte Regel lautet so:

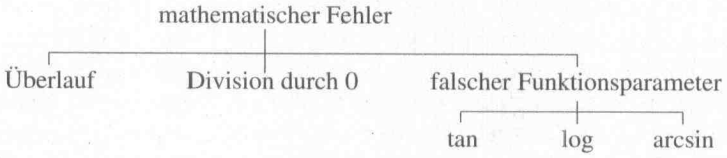
Eine Exception des Typs `"E"` kann von einer Catch-Anweisung des Typs `"C"`, `"const C"`, `"C&"` oder `"const C&"` aufgefangen werden, wenn

a) `C` und `E` identisch sind oder
b) `E` ein Zeigertyp und `C` mit `"void*"` identisch ist oder
c) `C` eine Basisklasse von `E` ist oder
d) `E` ein Zeiger auf eine Klasse `X` und `C` ein Zeiger auf eine Basisklasse von `X` ist.

Die Regel (**a**) beschreibt den trivialen Fall, den Sie schon in den bisherigen kleinen Beispielen gesehen haben, und nach (**b**) kann offensichtlich ein `"catch(void *v)"` Ausnahmen beliebiger Zeigertypen auffangen. Die beiden letzten Regeln sind wesentlich interessanter, denn sie ermöglichen die beliebte Technik, Ausnahmen zusammenzufassen.

Ein fast schon kanonisches Beispiel: Nehmen wir an, eine mathematische Bibliothek werfe verschiedene Ausnahmen. Denkbar wären z. B. der arithmetische Überlauf, Division durch Null und ein falsches Funktionsargument bei `"tan"`, `"log"` und `"arcsin"`. Für jeden dieser fünf Fehler kann man nun eine Fehlerklasse einführen und bei einer Exception einen Ausdruck der betreffenden Klasse werfen. Der Haken an der Sache ist, daß ein Programm, daß die Bibliothek benutzt, im Zweifelsfall immer fünf verschiedene Handler bereithalten müßte.

Andererseits gibt es im genannten Beispiel aber auch eine leicht sichtbare Hierarchie:



Also führt man, wenn man denn klug ist, die folgende Klassenhierarchie ein:

```
class MathErr { };

class OverflowErr: public MathErr { };
class DivisionErr: public MathErr { };
```

```
class FunctionErr: public MathErr { };

class TanParamErr: public FunctionErr { };
class LogParamErr: public FunctionErr { };
class ArcSinParamErr: public FunctionErr { };
```

Die Bibliothek wirft dann immer exakte Fehlermeldungen aus, z. B.

```
throw DivisionErr();
```

Man beachte, daß diese Klassenhierarchie aus lauter leeren (d. h. datenlosen) Klassen besteht, denn wir gehen einmal davon aus, daß außer dem Typ keine weiteren Informationen über den Laufzeitfehler benötigt werden. Die obige Anweisung mag Sie vielleicht verwirren: **"DivisionErr()"** ist kein Funktionsaufruf, sondern erzeugt mit Hilfe des Defaultkonstruktors ein namenloses temporäres Objekt eben dieser Klasse.

Der Trick an der Sache: Der Benutzer unserer Mathe-Bibliothek kann jetzt selbst einscheiden, ob er pauschal irgendwelche mathematischen Fehler oder ganz bestimmte Fehler abfangen will, oder er kann sogar mischen:

```
try
 {
   // Mathe-Aufrufe
 }
catch (DivisionErr &d)
 {
  cout << "Division durch Null!";
 }
catch (MathErr &m)
 {
  cout << "Mathematischer Fehler!";
 }
```

Der zweite Handler kann beliebige von **"MathErr"** abgeleitete Ausnahmen behandeln, der erste nur die der Klasse **"DivisionErr"** (oder auch Ausnahmen davon abgeleiteter Klassen, aber die gibt es hier ja nicht). Da der **"DivisionErr"**-Handler im Programmtext vor dem allgemeinen **"MathErr"**-Handler steht, wird er bei **"throw DivisionErr()"** auch zuerst gefunden und aufgerufen - bei umgekehrter Reihenfolge wäre er unerreichbar.

Damit Sie mal gesehen haben, daß es das auch gibt, wurden im obigen Beispiel in den Handlern Referenztypen benutzt. Die Wirkung ist eigentlich genau wie bei einem Funktionsaufruf: Der von der Exception geworfene Ausdruck wird dann nicht umkopiert, sondern nur in Form einer Adresse (also Referenz) an den Handler übergeben. In diesem Fall bringt das natürlich keinen Gewinn, denn das Umkopieren eines absolut leeren Klassenobjekts wie **"DivisionErr()"** kostet natürlich keinerlei Zeit.

Einen kleinen Unterschied gibt es aber doch. Betrachten Sie doch mal das folgende Beispiel:

```
class Base
 { public:
     virtual void f();
 };

class Derive: public Base
 { public:
     void f();
 };

void schmeiss()
 {
   throw Derive();
 }

void fang()
 {
  try
   {
    schmeiss();
   }
  catch (Base &b)
   {
    b.f();        // welche Funktion?
   }
 }
```

So, wie das Programm jetzt dasteht, erhält der Handler eine Referenz auf das geschmissene **"Derive"**-Objekt, und folglich wird die Funktion **"Derive::f"** aufgerufen. Würde man aber das kleine, unauffällige **"&"** in der Catch-Anweisung weglassen, wäre **"b"** ein Objekt der Klasse **"Base"**, und folglich käme **"Base::f"** zum Zuge.

Wenn Sie von solchen Spitzfindigkeiten einmal absehen, müssen Sie doch wohl zugeben, daß Fehler-Hierarchien eine feine Sache sind. Der nächste logische Schritt ist, sich eine Klasse zu definieren, von der alle überhaupt auftretenden Ausnahmeklassen abgeleitet werden. Kanonisch wäre es, diese Klasse **"Exception"** zu nennen. Da stoßen Sie allerdings auf das Problem, daß es noch keine Norm für Standard-Exceptions gibt. Der ANSI-C++-Standard, dessen Erscheinen wir vielleicht noch erleben werden, könnte dies tun. Bis dahin gilt die unverbindliche Empfehlung von C++-Papst Bjarne Stroustrup, diese Klasse **"Exception"** bis auf einen virtuellen Destruktor völlig leer zu lassen. Eine solche Klasse wird bei MaxonC++ in der Include-Datei „*<exception.h>*“ definiert und ist z. B. auch die Basis aller Ausnahmen, die von der MaxonC++ Klassenbibliothek geworfen werden.

8.4 Vom Fangen und wieder Wegwerfen

Ein paar ganz nützliche Variationen gibt es beim Auffangen von Ausnahmen noch und sollen Ihnen nicht vorenthalten werden. Zuerst sei kurz erwähnt, daß der Parameter eines Handlers auch namenlos sein kann, z. B.

```
catch (Matherr&)
  {
    // ...ganz normaler Code
  }
```

So kann man natürlich nicht auf das geworfene Datenobjekt zugreifen, aber meist ist dessen Klasse als Information schon vollkommen ausreichend. Sie können sich also die Mühe ersparen, sich einen Namen für einen unnützen Parameter auszudenken und einzutippen. Ansonsten funktioniert ein solcher „namenloser" Handler natürlich genau wie die bisher vorgestellten.

Es gibt nicht nur namenlose, sondern auch typlose Handler. Wenn eine Funktion beliebige Argumente haben darf, verwendet man bekanntlich eine Parameterliste mit Ellipse `"..."`. Diese Syntax ist auch bei Catch-Anweisungen erlaubt:

```
catch(...)
  {
    cout << "Da ist wohl was schiefgegangen";
  }
```

Dieser Handler fängt also jede beliebige Ausnahme auf. Innerhalb der Catch-Anweisung sind dann zwar keine Informationen über die Art der Ausnahme verfügbar, aber oft genug will man es gar nicht so genau wissen.

Ausnahmen können an wirklich jeder Stelle des Programms auftreten, und damit natürlich auch innerhalb einer Ausnahmebehandlung. Wenn es während der Ausführung einer Catch-Anweisung zu einer erneuten Exception kommt, wird die Catch-Anweisung natürlich abgebrochen und ein passender Handler für die neue Exception angesprungen. Dabei werden die Catch-Handler des gerade abgearbeiteten Try-Blocks nicht mehr in Betracht gezogen, denn die zugehörige Try-Anweisung wurde ja bereits mit der ersten Exception beendet.

Es ist sogar eine halbwegs gängige Praxis, in Handlern gezielt Ausnahmen zu werfen, wenn man feststellt, daß eine Ausnahme doch nicht behandelt werden kann. Ein Beispiel:

```
class DosFehler
{ public:
    int errorcode;
};

void irgendwas()
{
  try
    {
      // irgendwelche Dateioperationen
    }
```

```
    catch (DosFehler &df)
     {
       if (df.errorcode == 205)
         cout << "Tut mir leid, die Datei wurde nicht gefunden.";
       else
         throw df;
     }
}
```

Dieser Handler kümmert sich nur um den wohl harmlosesten Dateifehler, nämlich eine nicht vorhandene Datei. Im Amiga-DOS hat dieser Fehler den Code 205, weshalb übrigens auch in der Datei „*<dos/dos.h>*" die Konstante **"ERROR_OBJECT_NOT_FOUND"** entsprechend definiert ist. Bei allen anderen denkbaren Fehlerursachen betrachtet dieser Handler sich als unzuständig und gibt die Exception deshalb an einen hoffentlich existierenden anderen Handler weiter.

Das geht aber auch einfacher: Ein **"throw;"** ganz ohne Argument wirft die zuletzt aufgetretene und noch nicht vollständig behandelte Ausnahme noch einmal. Auch dazu ein Beispiel:

```
void kritisch()
{
  char *speicher = new char[50000];
  if (!speicher) throw MemError();

  try
   {
     mach_was(speicher); // hier passiert irgendetwas
   }
  catch(...)
   {
     delete [] speicher;
     throw;
   }

  delete [] speicher;
}
```

Ganz ohne Try-Block und Ausnahmebehandlung wäre diese Funktion in der Tat kritisch: Am Anfang wird ein nicht unerheblicher Speicherblock angefordert. Wenn nun aber in der Funktion **„mach_was"** eine Ausnahme aufträte, würde das Ende der Funktion **„kritisch"** und damit die **"delete"**-Anweisung nicht erreicht. Die 50 KByte Speicher würden also nicht wieder freigegeben. Auf einem PC, der mit den momentan verfügbaren Windows-Versionen ja unter 8 MByte RAM kaum noch lauffähig ist (Windows „Chicago" 95 war zur Drucklegung dieses Buchs noch nicht verfügbar, weshalb ich keine Prognose über den zukünftigen Speicherbedarf wage), ist der Anwender an Speicherverschwendung gewöhnt und würde ein solches Programm wohl achselzuckend hinnehmen. Der Amiga-User legt aber Wert auf intelligente - und damit auch speichersparend geschriebene - Software, weshalb **"kritisch()"** diesen auf den ersten Blick etwas seltsamen Exception Handler erhalten hat.

Das `"catch(...)"` fängt zunächst alle Probleme ab, die in `"mach_was()"` auftreten können. Im Handler wird zunächst der angeforderte Speicher wieder freigegeben. Da wir aber eigentlich nicht vorhatten, in gerade dieser Funktion eine detaillierte Ausnahmenbehandlung zu realisieren, werfen wir die aufgetretene Ausnahme mit `"throw;"` einfach wieder aus und überlassen sie den anderen (hoffentlich) gerade gültigen Handlern. Unser Handler nimmt damit keinen Einfluß auf den Ablauf des Programms und des Exception Handlings und sorgt lediglich dafür, daß der Speicher aufgeräumt wird.

Im folgenden Kapitel finden Sie Hinweise auf eine Methode, wie man das Problem auch eleganter lösen könnte. Gestatten Sie mir zunächst noch den Hinweis, daß das argumentlose `"throw;"` nicht nur direkt innerhalb einer Catch-Anweisung, sondern an jeder beliebigen Stelle eines Programms auftreten darf. Die folgende Funktion wäre z. B. legal:

```
void schmeissweg()
  {
    throw;
  }
```

Nun kann der Compiler nicht prüfen, wo genau diese Funktion später einmal aufgerufen wird. Es ist gut möglich, daß `"schmeissweg"` aufgerufen wird, ohne daß überhaupt eine Ausnahme aufgetreten wäre. Dann wird die Funktion `"abort"` aufgerufen, was zum sofortigen Abbruch des Programms führt. Man kann das getrost auch einen Laufzeitfehler nennen.

8.5 Von Konstruktoren und Destruktoren

Schon im guten alten ANSI C gabe es so etwas wie eine Ausnahmebehandlung, nämlich die beiden Funktionen `"setjmp"` und `"longjmp"`. Auch mit diesen simplen Funktionen konnte man auf einen Rutsch aus tief verschachtelten Funktionsaufrufen aussteigen, was sich besonders gut zur Behandlung kniffliger Laufzeitfehler eignete. Warum also braucht man noch das Exception Handling?

Zwei Vorteile sind bereits offensichtlich: Exception Handling ist übersichtlicher als diese obskuren `"setjmp"`-Konstrukte, und man kann für verschiedene Fehler auch verschiedene Handler angeben, ohne sich um die Auswahl des jeweils passenden Handlers näher zu kümmern.

Die C++ Ausnahmebehandlung hat aber noch eine weitere erfreuliche Eigenschaft: Destruktoren für lokale Variablen werden dabei korrekt aufgerufen. Ein Beispiel:

```
void mach_was(istream &is)
 {
   for (;;)                    // eigentlich eine Endlosschleife
    { if (!is) throw 42;       // verwegener Schleifenaussteig!
      cout << (char)is.get(); // Zeichen lesen und ausgeben
    }
 }

void fileop(char *name)
 {
```

```
    ifstream is(name);
    mach_was(is);
    cout << "Hierhin dürften wir kaum kommen...";
  }

int main()
 {
   try
    { fileop("s:startup-sequence"); }
   catch (int i)
    { cout << "Ausnahme " << i << " aufgetreten.\n"; }
 }
```

Bekanntlich wird beim Destruieren eines Stream-Objekts die zugehörige Datei geschlossen, sofern das noch nicht geschehen ist. Wenn hier in **"mach_was"** eine Ausnahme geworfen und diese in **"main"** aufgefangen wird, wird vorher noch das Objekt **"is"** aus der Funktion **"fileop"** destruiert und damit die Datei geschlossen.

Dieses Feature kann man wie versprochen benutzen, um das Beispiel aus Abschnitt 8.4 eleganter zu formulieren:

```
class MemError { };  // nur ein Dummy

class MemWatch
{   char *p;
  public:
    MemWatch(unsigned size)
      { if (!(p=new char[size])) throw MemError(); }

    ~MemWatch() { delete [] p; }

    operator char*() { return p; }
};

void mach_was(char*);

void kritisch()
{
  MemWatch speicher(50000);
  mach_was(speicher);
}
```

Die Klasse **"MemWatch"** realisiert einen Pointer mit automatischer Speicherfreigabe. Ganz egal, ob nun in **"mach_was"** eine Ausnahme auftritt oder nicht: Das Objekt **"speicher"** wird auf jeden Fall destruiert und damit der angeforderte Speicher freigegeben.

Bei der Konzeption der Ausnahmebehandlung von C++ ist man davon ausgegangen, daß eine Exception relativ selten auftritt. Wenn ein Teil eines Programms läuft, ohne daß irgendwelche Ausnahmen auftreten, soll es idealerweise genauso schnell und kompakt wie ein gleichwertiges Programm ganz ohne Exception Handling sein (dafür darf die Behandlung einer Ausnahme auch etwas

Zeit kosten). Dieses Ziel wird von MaxonC++ beinahe erreicht. Ich sage „beinahe", weil ein Programm trotzdem in irgendeiner Weise protokollieren muß, welche Objekte im Falle einer Exception wie destruiert werden müssen.

Daraus folgt, daß Programme mit Ausnahmebehandlung in MaxonC++ 3 doch etwas (aber nicht viel!) länger und langsamer geraten als bei den bisherigen Compiler-Versionen. Für Leute, die Exception Handling nicht mögen, gibt es deshalb die Möglichkeit, sie schlicht auszuschalten. Benutzt man die Option `"-pe"` oder den entsprechenden Schalter im Compiler-Dialogfenster, so schraubt man den Compiler auf einen „2.0 plus Templates"-Sprachstandard herunter. Die Schlüsselworte `"try"`, `"throw"` und `"catch"` werden nicht mehr als solche erkannt, und es wird kein Code mehr für die Destruktor-Buchführung erzeugt. Mein Tip lautet aber: Lassen Sie die Finger von diesem Schalter. Bei „klassischen" C-Programmen macht es überhaupt keinen und bei C++-Programmen nur einen geringen Unterschied, und für die MaxonC++ Klassenbibliothek brauchen Sie die Exceptions sowieso.

Zurück zum Thema Destruktoren: Eines der klassischen Probleme objektorienter Programmierung war die Frage, was man machen soll, wenn beim Aufruf eines Konstruktors ein Fehler auftritt. Bisher behalf man sich in C++ so, daß das Objekt dann nur „provisorisch" konstruiert wird und der Programmierer den Zustand manuell abfragen muß. Das bekannteste Beispiel ist wohl:

```
ifstream is(name);
if (!is) { ...Fehlerbehandlung... }
```

Schön ist so etwas natürlich nicht. Seit es Ausnahmebehandlung gibt, liegt es natürlich nahe, beim Scheitern eines Konstruktoraufrufs eine Exception auszuwerfen. Neben dem Wegfallen der manuellen Fehlerbehandlung hat das den Vorteil, daß nie ein unvollständig konstruiertes Objekt existieren kann. Natürlich wird dann auch der Destruktor für das un-konstruierte Objekt (hier also `"is"`) nie aufgerufen.

Was das ganze mit Destruktoren zu tun hat? Nun, ich möchte in ganz einfach darauf hinweisen, daß bei Konstruktoraufrufen alles so abläuft, wie man sich das wünscht:

```
class B1
{ public: B1(int);
          ~B1(); };

class B2
{ public: B2(char *);
          ~B2(); };

class B3
{ public: B3(double);
          ~B3(); };

class C : public B1, public B2, private B3
{
    B1 m1, m2, m3;
  public:
```

```
    C();
};

C::C() : B1(17), B2("vier"), B3(21.0), m1(26), m2(7), m3(31)
{
   // bis hierher kann schon beliebig viel schiefgegangen sein!
   throw 42;                          // ...und jetzt auch noch das!
}
```

Ignorieren Sie bitte zunächst einmal die Throw-Anweisung im Konstruktor von `"C"`. Prinzipiell können schon die Konstruktoren von `"B1"`, `"B2"` und `"B3"` beliebig viele Exceptions verursachen. Nehmen wir doch einmal an, bei der Initialisierung des Members `"m2"` tritt eben dies auf. Natürlich wird der Konstruktor von `"C"` an dieser Stelle abgebrochen und die Ausnahme an den aufrufenden Code weitergegeben. Das Problem dabei sind die drei Basisklassen sowie der Member `"m1"`, die ja alle schon initialisiert sind. Man stelle sich nur vor, eines dieser Objekte sei so etwas wie die Klasse `"MemWatch"` aus dem vorletzten Beispiel: Dann würde der angeforderte Speicherblock ja überhaupt nicht mehr freigegeben werden können! Und zu allem Überfluß ist es auch syntaktisch nicht möglich, um die Initialisierungsliste einen Try-Block zu legen und die Sache manuell zu erledigen.

Aber keine Angst: Konstruktoren und Destruktoren gehören in C++ zusammen wie Marianne Rosenberg und Brechreiz - das eine gibt es nie ohne das andere. Auch bei Ausnahmen in Konstruktoren bleibt C++ konsistent. Wenn während der Ausführung eines Konstruktors eine Exception auftritt, werden alle Basisklassen und Member, die zum betreffenden Zeitpunkt schon initialisiert sind, automatisch destruiert. Das gilt übrigens nicht nur für die Initialisierungsliste, sondern auch für Exceptions, die im eigentlichen Anweisungsteil des Konstruktors ausgeworfen werden. Im obigen Beispiel würden beim `"throw 42;"` also sämtliche Basisklassen und Member ganz von selbst destruiert.

8.6 Von erwarteten und unerwarteten Ausnahmen

Vielleicht gehören Sie zu jenem mißtrauischen Menschenschlag, der immer alles unter Kontrolle haben will. Dann dürfte Ihnen das Exception Handling in der bisher beschriebenen Form kaum gefallen: Sie benutzen eine Bibliotheksfunktion und haben keine Ahnung, ob diese nicht - Patsch! - plötzlich eine Ausnahme wirft und Ihr Programm abbricht. Das Problem kann man natürlich durch eine ausreichend detaillierte Dokumentation der Bibliothek lösen, aber zum einen sind Handbuchschreiber zuweilen äußerst nachlässige Zeitgenossen (da bin ich natürlich ausgenommen!) und zum anderen ist eine Klassenbibliothek, die man ohne ihren Quelltext erworben hat, manchmal eine genauso dubiose Angelegenheit wie ein Billig-PC von **b*s: Man weiß nie genau, was drinsteckt.

Der Gedanke liegt deshalb nahe, schon bei der Deklaration einer Funktion zu verraten, welche Ausnahmen diese einmal werfen soll. Das tut man, indem man hinter den Funktionsnamen `"throw"` schreibt und eine Liste von Datentypen folgen läßt, z. B.

```
class MemError{ };
class DOSError{ };

void f(int) throw (MemError,DOSError);
```

```
int g(char *a, char *b) throw(MemError)
{
  //...irgendwas...
  throw MemError();
}

class C
{ public:
    void Write() throw(DOSError);
    C(const char*) throw(MemError);
};

C::C(const char*)      // "throw" muß nicht wiederholt werden...
{ // ...
}

void Write() throw(DOSError)     // ...darf aber!
{ // ...
}

void h(const C&) throw();
```

In diesen Beispielen haben wir schon fast alles versammelt, was es zu diesen „Ausnahme-Deklarationen" zu sagen gibt. Falls Sie gerade verwirrt sein sollten: Nein, da stehen keine Throw-Anweisungen! Das Schlüsselwort `"throw"` wird hier zu einem ganz anderen Zweck eingesetzt. Nachdem schon `"static"` vier oder fünf verschiedene Bedeutungen hat, hatte man hier offensichtlich keinen Hemmungen, auch `"throw"` zu überladen.

Zunächst zur Bedeutung einer Ausnahme-Deklaration: Sie entspricht einer Garantie, daß die Funktion keine anderen als die angegebenen Exceptions auswirft. Eine Funktion wie

```
void f(int i) throw (MemError,DOSError)
{
  do_something();
}
```

kann natürlich wieder andere Funktionen aufrufen, und es kann dann passieren, daß diese Funktionen „unerlaubte" Ausnahmen auswerfen. Der Compiler kann deshalb nicht prüfen, ob eine Funktion wirklich das von der Ausnahme-Deklaration versprochene Verhalten zeigt.

Ist eine Ausnahme-Deklaration dann nichts anderes als ein unverbindlicher Kommentar? Keineswegs, denn die obige Funktion `"f"` entspricht

```
void f(int i)
{
  try
  {
    do_something();
  }
  catch(MemError)
```

```
    {
      throw;
    }
    catch(DOSError)
    {
      throw;
    }
    catch(...)
    {
      unexpected();
    }
}
```

Das heißt: Wenn eine Funktion eine Ausnahme-Deklaration besitzt, wird um ihren gesamten Anweisungsteil (bei Konstruktoren auch um die Initialisierungsliste, was sonst rein syntaktisch natürlich nicht geht) ein Try-Block gelegt, der „legale" Exceptions unverändert wieder auswirft und bei allen anderen eine Funktion namens **"unexpected"** aufruft, welche dann normalerweise **"abort"** aufruft und somit das Programm eiskalt abbricht.

Der Compiler kann also nicht kontrollieren, ob eine Funktion sich an ihre Ausnahme-Deklaration hält, aber er erzeugt immerhin Code, der andernfalls das Programm abbricht.

Noch ein paar Anmerkungen zur Syntax: Wenn man für eine Funktion einmal eine Ausnahme-Deklaration angegeben hat, kann man sie bei später nachfolgenden Deklarationen und Definitionen derselben Funktion wahlweise weglassen oder genau so wiederholen - das macht keinen Unterschied. Eine Funktion

```
void f() throw();
```

darf überhaupt keine Exceptions auswerfen, während eine „konventionelle" Funktion

```
void f();
```

jede beliebige Exception werfen darf.

Wenn in einer Ausnahme-Deklaration eine Klasse **"C"** aufgeführt wird, kann die Funktion auch Ausnahmen jeder von **"C"** abgeleiteten Klasse schmeißen. Für Zeiger auf Klassen gilt das entsprechend analog.

Zum Schluß noch ein paar Anmerkungen zur Bibliotheksfunktion **"unexpected"**: Ein Aufruf dieser Funktion sollte normalerweise nicht vorkommen, jedenfalls nicht mehr im fertigen Programm. In der Debuggingphase ist sie aber ganz nützlich. Normalerweise macht **"unexpected"** nichts anderes, als eine andere Funktion namens **"terminate"** aufzurufen, welche sich wiederum darauf beschränkt, **"abort"** anzuspringen. Man kann das aber ändern: Der Funktion **"set_unexpected"** kann man einen Zeiger auf eine andere parameterlose Funktion übergeben, die von **"unexpected"** aufgerufen werden soll, und liefert als Ergebnis die bisher gesetzte Funktion - beim ersten mal also **"terminate"**.

Wenn man mit einem kleinen `"typedef"` den in diesen Fällen üblichen Syntax-Kuddelmuddel von C umgeht, ist `"set_unexpected"` also wie folgt definiert:

```
typedef void (*PVF)();
PVF set_unexpected(PVF);
```

Analog dazu gibt es noch die Funktion `"set_terminate"`, mit der man festlegen kann, wohin `"terminate"` springen soll (Default ist ja `"abort"`). Alle diese Funktionen werden, zusammen mit der Klasse `"Exception"`, in der Includedatei „*<exceptions.h>*" deklariert.

8.7 Beispielprogramm: Das Übliche

Damit der Kreis sich schließt, möchte ich noch mal das Beispielprogramm aus Kapitel 6 ausgraben, in dem bekanntlich das gesammelte Know-How der Beispiele aus den Kapiteln 2 bis 4 steckt. Wir bringen also ein paar kleine Ergänzungen an, um unsere Listen endgültig zur Perfektion zu bringen.

Diese Ergänzungen sind simpler Natur: Wir definieren zuerst eine Exception, die ausgeworfen wird, wenn kein Speicher für neue Elemente mehr frei ist.

```
#include <exceptions.h>

class MemError : public Exception { };
```

Diese Exception wird dann von `"Liste::einfuegen"` geworfen:

```
template<class K, class D>
  void Liste<K,D>::einfuegen(const K &k, const D &d)
{
  // Position suchen:
  Listenelement<K,D> *neu, *pos = anfang;
  while (pos && pos->key > k)
     pos = pos->next;

  // neues Element erzeugen:
  neu = new Listenelement<K,D>;
  if (!neu) throw MemError();
  neu->key = k;
  neu->data= d;

  ...
```

Das war's dann auch schon. Tolles Beispielprogramm, nicht wahr?

AMIGA

MaxonC++

Referenz

MAXON
computer

1. Die ANSI C Includedateien

In diesem Abschnitt werden die Includedateien besprochen, die Bestandteil des ANSI C Standards sind. MaxonC++ deklariert darin aber auch einige zusätzliche eigene Funktionen und Symbole. Diese sind jeweils mit einem **„[M]"** gekennzeichnet.

1.1 Verschiedene nützliche Funktionen: <stdlib.h>

In der Include-Datei *<stdlib.h>* werden einige nützliche Funktionen für die unterschiedlichsten Zwecke, z. B. zur Umwandlung von Zahlen und für die Speicherverwaltung, definiert.

1.1.1 Umwandlung zwischen Zahlen und Strings

strtol

```
long int strtol(const char *string, char **end, int base)
```

Die Funktion `"strtol"` wandelt eine Zeichenkette in eine ganze Zahl um. Am Anfang von `"string"` darf Leerraum stehen und hinter der Zahldarstellung dürfen noch andere Zeichen folgen. Wird für `"end"` ein Wert ungleich Null angegeben, so wird ein Zeiger auf den nicht umgewandelten Teil der Zeichenkette (z. B. auf die Zeichen, die der Zahldarstellung folgen) in der Variablen, auf die `"end"` verweist, abgelegt. Der Parameter `"base"` gibt die gewünschte Basis an: bei einem Wert zwischen 2 und 36 wird die Zeichenkette mit dieser Basis umgewandelt; ist der Wert 0, hängt die Basis wie in C üblich vom Präfix der Zeichenfolge ab (Hex bei `"0x"` oder `"0X"`, oktal bei `"0"` und andernfalls dezimal).

Wenn das Ergebnis den Bereich von `"long int"` über- oder unterschreitet, wird `"LONG_MAX"` bzw. `"LONG_MIN"` zurückgegeben und die Variable `"errno"` erhält den Wert `"ERANGE"` (siehe *<errno.h>*).

strtoul

```
unsigned long strtoul(const char *string, char **end, int base)
```

Diese Funktion ist analog zu `"strtol"`, aber der String wird in ein vorzeichenloses Langwort umgewandelt und im Falle eines Fehlers wird die Konstante `"ULONG_MAX"` zurückgegeben.

strtod

```
double strtod(const char *string, char **end)
```

Auch diese Funktion wandelt den Anfang einer Zeichenkette in eine Zahl um und zwar in einen "double"-Wert. Wie bei "strtol" wird ein Zeiger auf den nicht umgewandelten Teil der Zeichenkette in "*end" abgelegt, sofern "end" nicht Null ist. Bei einem Überlauf ist das Ergebnis "HUGE_VAL", bei einem Unterlauf wird "0" zurückgegeben.

strtovl [M]

```
long long strtovl(const char *string, char **end, int base)
```

Diese Funktion entspricht "strtol", wandelt aber in einen 64 Bit breiten "long long int"-Wert um, und im Falle eines Überlaufs ist das Ergebnis "LONGLONG_MAX" bzw. "LONGLONG_MIN".

strtouvl [M]

```
unsigned long long strtouvl(const char *s, char **end, int base)
```

Auch "strtouvl" führt in gewohnter Weise die Umwandlung eines Strings durch und zwar nach "unsigned long long int". Beim Überlauf wird "ULONGLONG_MAX" als Ergebnis geliefert, sonst ist alles wie bei "strtol".

atoi

```
int atoi(const char *string)
```

"atoi" wandelt eine dezimale Ziffernfolge in den entsprechenden numerischen Wert um und entspricht damit "(int) strtol(string, 0, 10)".

atol

```
long atol(const char *string)
```

Auch diese Funktion ist nur eine vereinfachte Form von "strtol" und entspricht "strtol(string, 0, 10)".

atof

```
double atof(const char *string)
```

Noch eine Kurzform, diesmal aber im Fileßkommabereich: Ein Aufruf von `"atof"` ist äquivalent zu `"strtod(string, 0)"`.

ERANGE

```
#define ERANGE 1000
```

Diesen Wert erhält die Variable `"errno"`, wenn bei `"strtol"` oder den verwandten Funktionen ein Überlauf auftritt.

HUGE_VAL

```
#define HUGE_VAL 1.797693134862316E+308
```

Rückgabewert der Funktion `"strtod"` im Falle eines Überlaufs.

1.1.2 Mathematische Funktionen

abs

```
int abs(int i)
```

`"abs"` liefert den absoluten Betrag seines Arguments, also so etwas wie `"(i >= 0? +i : -i)"`.

labs

```
long int labs(long int l)
```

Genau wie `"abs"`, nur eben für `"long int"`-Werte. Bekanntlich sind in MaxonC++ die internen Darstellungen von `"int"` und `"long"` identisch (jeweils 32 Bit), so daß auch diese beiden Funktionen sich praktisch nicht unterscheiden.

vlabs [M]

```
long long int vlabs(long long int vl)
```

Noch eine Betragsfunktion, diesmal für `"long long int"` und damit natürlich eine Spezialität von MaxonC++. Ganz schön nervig, daß man in ANSI C nicht überladen kann...

div

```
div_t div(int nenner, int zaehler)
```

Da bei den üblichen Divisionsalgorithmen gleichzeitig Quotient und Rest ermittelt werden, ist es natürlich Zeitverschwendung, diese beiden Werte in zwei getrennten Rechenschritten zu berechnen. Der Datentyp `"div_t"` ist eine Struktur mit den Einträgen `"quot"` für den Quotienten und `"rem"` für den Rest.

ldiv_t

```
ldiv_t ldiv(long nenner, long zaehler)
```

Das Ganze noch einmal für `"long"`-Zahlen.

div_t

```
typedef struct {
     int quot;
     int rem;
     } div_t
```

In Strukturen dieses Typs werden die Ergebnisse der Funktion `"div"` zurückgegeben.

ldiv_t

```
typedef struct {
     long quot;
     long rem;
     } ldiv_t;
```

ist die Ergebnis-Struktur der Funktion `"ldiv"`.

1.1.3 Zufallszahlen

RAND_MAX

```
#define RAND_MAX 0x7fff
```

Diese Konstante stellt das größtmögliche Ergebnis von `"rand"` dar.

rand

```
int rand( void )
```

`"rand"` liefert eine Pseudo-Zufallszahl im Bereich von 0 bis RAND_MAX. MaxonC^{++} verwendet hier das additive Kongruenzverfahren, so daß ein Startwert definiert werden muß und zwar mit der folgenden Funktion:

srand

```
void srand(unsigned int seed)
```

Da die wenigsten Amigas einen Hardware-Zufallszahlengenerator eingebaut haben (am besten solche Dinger, die den Zerfall von Isotopen benutzen, denn solche Vorgänge sind wirklich zufällig), berechnet `"rand"` natürlich nur Pseudo-Zufallszahlen nach einem bestimmten Algorithmus (s. o.). Dieser Algorithmus benötigt einen Startwert, aus dem die erste Zufallszahl berechnet wird (und dann jede weitere Zahl aus der vorhergehenden). `"srand"` setzt diesen Startwert und sollte deshalb vor der ersten Benutzung von `"rand"` aufgerufen werden.

Wenn Sie dabei einen konstanten Wert verwenden, etwa `"srand(4711)"`, erhalten Sie natürlich auch immer dieselbe Folge von „Zufallszahlen". Deshalb ist es üblich, hier einen Wert zu wählen, der von der Systemzeit abhängt, z. B. `"srand(time(0))"`.

Random [M]

```
unsigned Random(unsigned u)
double Random()
```

Dieser überladene Funktionsname ist eine Spezialität von MaxonC^{++} und kann auch nur benutz werden, wenn *<stdlib.h>* im C^{++}-Modus compiliert wird. Mit einen ganzzahligen Argument `"u"` liefert `"Random"` eine Zufallszahl im Bereich von 0 bis u-1, ohne Argument eine Fließkommazahl zwischen 0 und 1 (wobei 0 möglicherweise vorkommen kann, 1 aber nicht). Diese Funk-

tionen benutzen ein ähnliches Verfahren wie `"rand"`, beziehen aber die aktuelle Rasterzeile des Videoausgangs in die Berechnung mit ein. Dadurch muß die Zufallszahlenfolge nicht initialisiert werden und ist deshalb wesentlich „zufälliger" als `"rand"`.

1.1.4 Speicherverwaltung, Typen und Zeiger

malloc

```
void malloc(size_t size)
```

`"malloc"` reserviert `"size"` Bytes Speicher und gibt einen Zeiger darauf zurück oder 0, wenn kein Speicher angefordert werden konnte. `"size_t"` ist ein ganzzahliger Datentyp, der unter MaxonC++ als `"unsigned"` definiert ist und „zufällig" auch der Typ des Ergebnisses von „sizeof" ist.

Mit der Funktion `"set_new_handler"` (siehe *<new.h>*) kann eine Funktion spezifiziert werden, die so lange immer wieder aufgerufen wird, bis genügend Speicher frei ist - oder die Funktion mit `"exit"` o. Ä. aussteigt.

calloc

```
void *calloc(size_t anzahl, size_t size)
```

Diese Funktion entspricht `"malloc"`, reserviert aber Speicher für einen „anzahl" Elemente großen Vektor des Elementtyps `"size"`.

free

```
void free(void *ptr)
```

`"free"` gibt einen mit `"malloc"`, `"calloc"` oder `"realloc"` reservierten Speicherbereich wieder frei.

realloc

```
void *realloc(void *ptr, size_t size)
```

Der Speicherplatzbedarf eines Objekts, auf das `"ptr"` zeigt, wird auf `"size"` geändert und ein Zeiger auf das (möglicherweise verschobene) Objekt zurückgegeben. Natürlich muß es sich um ein Objekt handeln, das mit `"malloc"`, `"calloc"` oder einem anderen `"realloc"`-Aufruf erzeugt wurde.

MaxonC^{++} verfährt dabei wie folgt:

- Ist die neue Größe mit der alten identisch, muß nichts getan werden, und der Zeiger `"ptr"` wird unverändert zurückgegeben.
- Andernfalls wird versucht, einen neuen Speicherbereich der gewünschten Größe zu allozieren, und die passende Anzahl von Bytes umkopiert. Das alte Objekt wird gelöscht und ein Zeiger auf das neue zurückgegeben.
- Kann kein neues Objekt eingerichtet werden, ist das Ergebnis 0, und das ursprüngliche Objekt, auf das `"ptr"` zeigt, bleibt unverändert.

In der ersten Version wurde noch bei Verkleinerung ein Teil des ursprünglichen Objekts freigegeben bzw. bei Vergrößerung versucht, den unmittelbar angrenzenden Speicher anzufordern. Das war zwar effektiv, widerspricht aber den Amiga-Programmierrichtlinien und bereitet insbesondere Tools wie „*MemWall*“ Probleme.

size_t

```
typedef unsigned size_t
```

`"size_t"` ist der Datentyp, der in C und C^{++} überall da verwendet wird, wo es um die Größe irgendwelcher Objekte geht.

NULL

```
#define NULL 0
```

Eine Konstante, die manche Leute gern für Zeiger des entsprechenden Werts verwenden. Eine Definition `"((void*)0)"` wie in vielen anderen C-Systemen ist hier übrigens nicht möglich, da C^{++} keine direkte Konvertierung von `"void*"` in andere Zeigertypen kennt.

wchar_t

```
typedef int wchar_t
```

Der Datentyp von „langen" Zeichenkonstanten, z. B.

```
wchar_t C = L'x';
```

1.1.5 Programmende

exit

```
void exit(int res)
```

Diese Funktion beendet das Programm und entspricht einem `"return res"` aus der Funktion `"int main()"`. Mit `"atexit"` eingehängte Funktione werden ausgeführt, Destruktoren für globale Variablen aufgerufen und Resourcen (Dateien, Speicher usw.) freigegeben. Das Ergebnis Null zeigt normalerweise an, daß das Programm erfolgreich ausgeführt wurde, und jeder andere Wert wird als Fehlernummer interpretiert.

atexit

```
int atexit(void (*funktion)(void))
```

teilt der Laufzeitbibliothek mit, daß die angegebene parameterlose Funktion am Programmende ausgeführt werden soll.

abort

```
void abort(void)
```

Das Programm wird gnadenlos abgebrochen, ganz ohne Destruktoren, `"atexit"`-Funktionen und Resourcen-Freigabe.

1.1.6 Kommunikation mit dem Betriebssystem

getenv

```
char *getenv(const char *name)
```

Die Funktion liefert den Wert der Environment-Variablen `"name"` und liefert einen Zeiger darauf oder Null, wenn keine derartige Variable existiert.

Diese Funktion benötigt mindestens die Kickstart-Version 2.0 (37.175) und liefert bei früheren Versionen stets Null.

system

```
int system(const char *command)
```

Die Funktion führt den in `"command"` beschriebenen CLI-Befehl aus und liefert dessen Rückgabewert als Ergebnis. Auch diese Funktion benötigt mindestens Betriebssystemversion 2.0 und liefert andernfalls immer `"-1"`. Die Ausgaben der Funktion erfolgen über die Standardausgabe des Prozesses - und in der Entwicklungsumgebung MCPP ist dies bei Betriebssystemversion 1.2 oder 1.3 leider die Umgebung, aus der MaxonC++ einmal gestartet wurde, nämlich das entsprechende CLI-Fenster oder `"Nil:"` beim Start von der Workbench. Ab Betriebssystemversion 2.0 werden die Ausgaben korrekt umgelenkt.

1.1.7 Sortieren und Suchen

qsort

```
void qsort(void *vektor, size_t anzahl, size_t size,
int (*compare)(const void *, const void *))
```

Diese Funktion sortiert den Vektor `"vektor[0] ... vektor[anzahl-1]"`, dessen Elemente jeweils die Größe `"size"` haben, mit der angegebenen Vergleichsfunktion. In Abschnitt 2.3.4 des Tutorials wird `"qsort"` ausführlich beschrieben.

bsearch

```
void *bsearch(const void *obj, const void *vektor,
  size_t anzahl, size_t size,
  int (*compare)(const void *, const void *))
```

sucht im Vektor `"vektor[0] ... vektor[anzahl-1]"`, dessen Elemente jeweils die Größe `"size"` haben, anhand der angegebenen Vergleichsfunktion ein Element, das mit `"*obj"` übereinstimmt. Da die Suche binär erfolgt, muß der Vektor dafür (z. B. mit `"qsort"`) sortiert sein. Auch diese Funktion wird in Abschnitt 2.3.4 des Tutorials näher erläutert.

1.2 Ein- und Ausgaben: <stdio.h>

1.2.1 Typen, Objekte und Konstanten

FILE

```
typedef struct stream FILE
```

Der Datenstrom ist das Hauptkonzept der Ein- und Ausgabe in ANSI C und entspricht einer Datei, die aber nicht unbedingt wie eine Datei aussehen (also auf einer Diskette oder Festplatte liegen) muß, sondern auch z. B. ein Terminal, ein Window oder ein Drucker sein kann. Der Datentyp, der solche Strom-Objekte beschreibt, heißt `"struct stream"` oder `"FILE"`.

std__in, std__out, std__err [M]

```
extern FILE std__in, std__out, std__err
```

Dies sind die Namen der drei Standard-Dateien: `"std__in"` für Eingaben des Programms, `"std__out"` für Ausgaben und `"std__err"` für Fehlermeldungen. Auf dem Amiga sind der Ausgabe- und der Fehlerkanal (leider) identisch. Beim Programmstart sind diese Dateien geöffnet und initialisiert (auch wenn sie beim Workbench-Start ins Nirvana zeigen) und müssen am Programmende auch nicht geschlossen werden.

Die drei Datenströme müssen keineswegs in jedem C-System diese Namen tragen, sondern können durchaus anders heißen. Deshalb sollte man sie nie unter diesen Bezeichnern ansprechen, sondern nur mit den drei folgenden Makros:

stdin, stdout, stderr

```
#define stdin (&std__in)
#define stdout (&std__out)
#define stderr (&std__err)
```

Die meisten Funktionen, die etwas mit Ein- und Ausgabe von Daten zu tun haben, erwarten als Argument keinen Datenstrom, sondern einen Zeiger auf einen solchen. Deshalb gibt es diese drei Makros, die jeweils einen Zeiger auf eine Standard-Datei liefern.

NULL

```
#define NULL 0
```

Die allseits beliebte Konstante (für Zeiger) wird auch in dieser Include-Datei definiert.

size_t

```
typedef unsigned size_t
```

`"size_t"` ist der Standardname für den Typ, den Ergebnisse von `"sizeof"` haben.

EOF

```
#define EOF (-1)
```

Die Konstante `"EOF"` wird an verschiedenen Stellen zum Anzeigen des Dateiendes bzw. eines Fehlers benutzt.

1.2.2 Öffnen und Schließen von Dateien

fopen

```
FILE *fopen(const char *name, const char *modus)
```

öffnet die Datei des angegebenen Namens und mit dem angegebenen Modus, erzeugt dazu eine FILE-Struktur und gibt einen Zeiger darauf zurück. Kann die Datei nicht geöffnet werden, liefert die Fuktion Null.

Es gibt die folgenden Modi:

`"r"` Vorhandene Datei zum Lesen öffnen.

`"w"` Datei zum Schreiben öffnen bzw. erzeugen und dabei evtl. vorhandene Datei überschreiben.

`"a"` Existierende Datei zum Anhängen ans Dateiende öffnen bzw. neue Datei erzeugen und zum Schreiben öffnen.

`"r+"` Vorhandene Datei zum Lesen und Schreiben öffnen oder ggf. eine neue Datei erzeugen.

`"w+"` Eine neue Datei erzeugen (ggf. existierende Datei löschen) und zum Lesen und Schreiben öffnen.

`"a+"` Vorhandene Datei fürs Lesen und Schreiben öffnen (ggf. neue Datei anlegen), wobei Schreib-/Leseposition auf Dateiende gesetzt wird.

ANSI C unterscheidet zwischen Text- und Binärdateien. Die oben aufgeführten Modi öffnen eine Datei als Textdatei. Will man eine Binärdatei öffnen, ist ein `"b"` an die zweite oder dritte Stelle des Modus-Strings zu setzen, z. B. `"rb"`, `"rb+"` oder `"r+b"`.

Auf dem Amiga (wie auch unter UNIX) sind die beiden Dateimodi identisch, und so ist es egal, ob Sie das zusätzliche `"b"` angeben oder nicht. MS-DOS Computerund die MessyDOS-Kopie Atari ST dagegen müssen tricksen und bei Textdateien dem Programm vorgaukeln, daß die Folge `"Linefeed+Return"` aus nur einem einzigen Zeichen besteht.

fclose

```
int fclose(FILE *f)
```

Eine geöffnete Datei muß auch wieder geschlossen werden. Die Funktion `"fclose"` schließt eine Datei, die mit `"fopen"` geöffnet wurde, und gibt die zugehörige `"FILE"`-Struktur frei. Vorher werden natürlich alle Puffer geleert (wie bei `"fflush"`). Das Ergebnis ist Null oder

`"EOF"` bei einem Fehler, aber auf dem Amiga kann beim Schließen einer Datei nichts schiefgehen.

freopen

```
FILE *freopen(const char *name, const char *modus, FILE *file)
```

leitet einen bereits geöffneten Datenstrom in eine andere Datei um. Die physikalische Datei, mit der `"file"` bisher verbunden war, wird geschlossen, eine neue Datei mit dem angegebenen Namen und Modus geöffnet und ein Verweis darauf in die vorhandene FILE-Struktur gepackt.

Diese Funktion dient vor allem dazu, die Standard-Datenströme `"stdin"`, `"stdout"` oder `"stderr"` in eine andere physikalische Datei umzuleiten.

FILENAME_MAX

```
#define FILENAME_MAX 200
```

Der Amiga kennt es bekanntlich so gut wie keine Beschränkung der Länge von Dateinamen, aber genau diese maximale Länge soll `"FILENAME_MAX"` eigentlich angeben. Die Konstante wurde mehr oder weniger willkürlich mit 200 definiert, denn länger wird ein Dateiname einschließlich Pfad wohl kaum werden.

FOPEN_MAX

```
#define FOPEN_MAX 99999
```

Dank dynamischer Datenorganisation ist die Anzahl der Dateien, die Sie mit MaxonC^{++} öffnen dürfen, nicht begrenzt und die Konstante `"FOPEN_MAX"` deshalb ziemlich Banane. Begrenzt ist diese Anzahl lediglich dadurch, daß irgendwann der Speicher voll ist oder das Betriebssystem sich sonstwie beschwert.

1.2.3 Strings und Zeichenketten

fgetc

```
int fgetc(FILE *f)
```

liest ein Zeichen aus der Datei `"f"` und wandelt es zuerst in `"unsigned char"` und dann in `"int"` um, denn bekanntlich gibt es in C keine klare Trennung zwischen Zahlen und Zeichen. Wenn das Dateiende überschritten wurde oder sonst ein Fehler auftrat, wird `"EOF"` geliefert.

<$igetc

```
int getc(FILE *f)
```

Diese Funktion ist witzigerweise mit `"fgetc"` identisch, allerdings dürfen Implementationen sie aus nicht nachvollziehbaren Gründen in Form eines Makros definieren. Wie erst im Schwinden der Dominanz des Signifikats die semantische Struktur der Horizontflucht, des steten Aufschubs der Signifikation, entsteht, so wird andererseits mit der Substitution des Signifikats durch den Funktionszusammenhang der Signifikanten die Horizonterfahrung als Motorik des ständigen Verweisens ungültig.

getchar

```
int getchar (void)
```

ist äquivalent zu `"getc(stdin)"`, liest also ein Zeichen aus der Standard-Eingabe.

ungetc

```
int ungetc(int c, FILE *f)
```

Manchmal merkt man erst, daß man genug gelesen hat, wenn man zuviel gelesen hat. Die Funktion `"ungetc"` stellt ein bereits gelesenes Zeichen in eine Eingabedatei zurück, so daß es beim nächsten Lesevorgang erneut gelesen wird. Bei einer „echten" Datei kann man sich `"ungetc"` wie das Zurücksetzten des Lesezeigers um ein Zeichen vorstellen, aber bei einer interaktiven Eingabe geht das so natürlich nicht. Deshalb hat jeder Datenstrom einen Puffer, in dem solche „zurückgestellten" Zeichen aufgenommen werden. Es paßt aber immer nur genau ein Zeichen in diesen Puffer.

fgets

```
char *fgets(char *s, int n, FILE *f)
```

"fgets" liest eine Zeile mit höchstens "n-1" Zeichen aus "f" und legt sie einschließlich eines abschließenden Null-Bytes im String "s" ab. Als Zeilenende gilt entweder ein Linefeed-Zeichen ('\n') oder das Dateiende (EOF). In beiden Fällen wird das jeweilige Zeichen aber nicht in die Stringvariable aufgenommen. Wurde das Dateiende bereits überschritten oder trat ein Fehler auf, gibt "fgets" Null, andernfalls "s" zurück.

gets

```
char *gets(char *s)
```

entspricht "fgets(s, STREAM_MAXSTRING, stdin)", liest also eine Zeile von der Standardeingabe.

STREAM_MAXSTRING [M]

```
#define STREAM_MAXSTRING 80
```

Diese Konstante gibt an wieviele Zeichen maximal von "gets" gelesen werden, ist aber nicht Bestandteil des ANSI C-Standards.

fputc

```
int fputc(int c, FILE *f)
```

Natürlich kann man nicht nur Daten lesen, sondern sie auch schreiben. "fputc" ist die wohl rudimentärste Ausgabefunktion und schreibt ein Zeichen "c" in eine Datei "f". Das Ergebnis ist "EOF" bei Fehler oder andernfalls das ausgegebene Zeichen.

putc

```
int putc(int c, FILE *f)
```

Ähnlich wie "getc" und "fgetc", ist diese Funktion mit "fputc" identisch, darf aber als Makro definiert sein. Sie wissen ja, erst im Schwinden der Dominanz des Signifikats entsteht...

putchar

```
int putchar(int c)
```

schreibt das Zeichen in die Standard-Ausgabe und ist damit identisch mit: `"putc(c, stdout)"`.

fputs

```
int fputs(const char *s, FILE *f)
```

Diese Funktion schreibt eine Zeichenfolge, die wie üblich mit einen Null-Zeichen enden muß, in die Datei `"f"`. Anstelle des abschließendne Nullbytes wird ein Zeilenvorschub, also ein `"\n"`, geschrieben.

puts

```
int puts(const char *s)
```

dient ebenfalls zur Ausgabe von Zeichenketten und ist äquivalent zu `"fputs(s, stdout)"`.

1.2.4 Formatierte Ausgabe

printf

```
int printf(const char *format, ...)
```

Diese ungemein nützliche Funktion ist die Standard-Ausgabefunktion in C, jedenfalls für Textdateien. Im Prinzip wird dabei die Zeichenkette `"format"` ausgegeben, nur an bestimmten Stellen werden andere Daten eingesetzt. Diese „bestimmten Stellen" sind Umwandlungsanweisungen, die stets mit einem `"%"` anfangen, mit einem Buchstaben enden und dazwischen noch zusätzliche Argumente haben dürfen. Ein Beispiel:

```
int a=17, b = 4;
printf("Die Summe von %d und %d ist %d.\n", a, b, a+b);
```

Die Zeichenfolge `"%d"` weist `"printf"` an, aus den sonstigen Argumenten das nächste zu nehmen, als „int" zu interpretieren und dezimal auszugeben. Also erzeugt der obige Funktionsaufruf folgende Ausgabe:

Die Summe von 17 und 4 ist 21.

Natürlich gibt es nicht nur `"%d"`, sondern eine ganze Fülle von Format-Anweisungen, die folgendes in der genannten Reihenfolge enthalten können oder müssen:

- Das erste Zeichen ist, wie bereits erwähnt, stets `"%"`.
- Eine optionale Folge von Flags in beliebiger Reihenfolge: Ein `"-"` bewirkt eine linksbündige Ausgabe der Daten (nur sinnvoll, wenn eine Feldbreite angegeben wird, siehe unten), `"+"` sorgt dafür, daß eine Zahl auf jeden Fall mit Vorzeichen (also ggf. `"+"`) ausgegeben wird, wogegen ein Leerzeichen veranlaßt, daß bei vorzeichenlosen Zahlen ein Leerzeichen an den Anfang gesetzt wird. Das Flag `"0"` füllt bei Zahlen-Ausgaben das Ausgabefeld mit Nullen statt mit Leerzeichen auf (ebenfalls nur sinnvoll, wenn Feldbreite gesetzt), und `"#"` hat bei einigen Ausgabeformaten eine besondere Wirkung: Hexadezimalen Ausgaben wird ein `"0x"`, oktalen eine `"0"` vorangestellt, Fließkommaausgaben enthalten auf jeden Fall einen Dezimalpunkt und bei den Anweisungen `"g"` und `"G"` werden Nullen am Ende nicht unterdrückt. Welch' ein Satz!
- Die Feldbreite und zwar als dezimale Ziffernfolge. Die Ausgabe des umgewandelten Arguments erfolgt (sofern nicht anders spezifiziert) linksbündig in einem Feld, das mindestens diese Breite hat, wobei bei Bedarf mit Leerzeichen (oder Nullen, siehe oben) aufgefüllt wird.
- Die Genauigkeit, die mit einem `"."` anfängt (damit `"printf"` sie von der Feldbreite unterscheiden kann), ebenfalls als dezimale Ziffernfolge. Bei Zeichenketten legt dies die maximale Anzahl von Zeichen fest, bei ganzzahligen Ausgaben die minimale Anzahl von Ziffern (nach links wird dann mit Nullen aufgefüllt) und bei Fließkommazahlen die Anzahl der Dezimalstellen.
- Ein Buchstabe als Längenangabe: Bei `"h"` ist das Argument als `"short"` oder `"unsigned short"` zu betrachten, `"l"` weist es als `"long int"` aus und bei `"L"` ist es entweder `"long double"` oder `"long long"` oder `"unsigned long long"`.
- Das Umwandlungszeichen, z. B. `"d"` für ganzzahlige dezimale Ausgaben.

Außer dem `"%"` am Anfang und dem Umwandlungszeichen am Ende sind alle genannten Argumente optional. Als Genauigkeit oder Feldbreite kann anstelle einer Ziffernfolge `"*"` gesetzt werden. In diesem Fall steht der fehlende numerische Wert als nächstes `"int"`-Argument in der Argumentliste.

Es gibt folgende Umwandlungszeichen:

Zeichen	Argumenttyp	Art der Ausgabe
`d`	int	Dezimal
`i`	int	Wie `"d"`
`o`	int, unsigned	Vorzeichenlos oktal, ohne `"0"` am Anfang
`x`	int, unsigned	Vorzeichenlos hexadezimal ohne führendes `"0x"` und mit kleinen Buchstaben
`X`	int, unsigned	Wie `"x"`, aber mit Großbuchstaben
`u`	unsigned	Dezimal ohne Vorzeichen
`c`	int	Einzelnes Zeichen als `"unsigned char"`
`s`	char*	String, der mit Nullbyte endet
`f`	double	Fließkomma-Ausgabe in der Form `"[-]xxx.yyy"`. Die Genauigkeit legt die Anzahl der Nachkommastellen fest, Default ist 6.
`e`	double	Exponentielle Darstellung `"x.yyyyyeõzzz"`, wobei die Genauigkeit wieder die Zahl der Nachkommastellen angibt (Default 6).
`E`	double	wie `"e"`, aber mit großen `"E"`: `"x.yyyyyEõzzz"`
`g`	double	Bei Argumenten, deren Exponent zwischen der Genauigkeit (z. B. 6 als Default) und -4 liegt, erfolgt die Ausgabe wie bei `"f"`, andernfalls wie bei `"e"`.
`G`	double	Entspricht `"g"`, aber es wird zwischen `"f"` und `"E"` gewählt.
`p`	void*	Hexadezimale Speicheradresse
`n`	int*	Bewirkt keine Ausgabe, sondern legt die Anzahl der bisher ausgegebenen Zeichen in der Variablen, auf die das Argument zeigt, ab.
`%`	-	Gibt ein Prozentzeichen aus.

Das Ergebnis von `"printf"` ist die Anzahl der ausgegebenen Zeichen oder negativ, wenn ein Fehler auftrat. Bitte denken Sie stets daran, daß `"printf"` eine Funktion ohne Netz und doppelten Boden ist und fehlerhafte Argumente bestenfalls zu chaotischen Ausgaben und schlimmstenfalls zu einem Systemabsturz führen können.

fprintf

```
int fprintf(FILE *f, const char *format, ...)
```

Diese Funktion arbeitet eigentlich genau wie `"printf"`, aber die Daten werden nicht über den Standard-Ausgabestrom, sondern in die Datei `"f"` ausgegeben. Andersrum formuliert, ist `"printf(irgendwas)"` eine Abkürzung für `"fprintf(stdout, irgendwas)"`.

sprintf

```
int sprintf(char *s, const char *format, ...)
```

Auch diese Funktion entspricht `"printf"`, aber die Ausgaben werden nirgendwo wirklich ausgegeben, sondern im String `"s"` abgelegt und mit einem Nullzeichen abgeschlossen. Sie als Programmierer sind selbst dafür verantwortlich, daß der Ziel-String groß genug ist, denn andernfalls wird (wie in C nicht anders zu erwarten) gnadenlos über den Vektor hinaus geschrieben, und der Guru gerät ins Meditieren.

vprintf, vfprintf, vsprintf

```
typedef unsigned va_list;

int vprintf(const char *format, va_list arg);

int vfprintf(FILE *f, const char *format, va_list arg);

int vsprintf(char *s, const char *format, va_list arg);
```

Diese Funktionen sind absolut äquivalent zu ihren Entsprechungen ohne das `"v"` am Anfang, aber die Argumente, die sonst dem Formatstring folgen, werden hier durch eine variable Argumentenliste (`"va_list"`) übergeben. Bitte lesen Sie dazu auch das Kapitel über *<stdarg.h>* oder fragen Sie Ihren Arzt oder Apotheker.

1.2.5 Formatierte Eingabe

scanf

```
int scanf(const char *format, ...)
```

Was `"printf"` für die Ausgabe, ist `"scanf"` für die Eingabe. Der Formatstring ist ähnlich aufgebaut wie bei `"printf"`, hat aber eine andere Bedeutung:

Leerzeichen oder Tabulatoren in der Formatzeichenkette werden ignoriert. Bei allen anderen Zeichen außer `"%"` erwartet `"scanf"`, daß dieses Zeichen als nächstes in der Eingabe folgt (eventuell mit vorangehenden Leerzeichen) und liest es bzw. bricht ab, wenn das Zeichen nicht folgt. Wieder leitet `"%"` Umwandlungsanweisungen ein.

Einem `"%"` kann optional ein `"*"` folgen (dann werden zwar Zeichen gelesen und umgewandelt, aber nirgendwo abgelegt), oder eine Zahl, die die maximale Feldbreite festlegt, also die maximale Anzahl von Zeichen, die bei dieser Umwandlung gelesen werden sollen. Die optionalen Zeichen `"h"`, `"l"` und `"L"` legen wie bei `"printf"` die Datenbreite fest.

Zu jeder Umwandlungsanweisung gehört in der restlichen Argumentliste ein Zeiger auf eine Variable, es sei denn, hinter dem `"%"` folgt ein `"*"` (siehe oben). Bei einer Umwandlung überliest `"scanf"` zunächst alle Zwischenraumzeichen einschließlich Zeilentrennern, liest eine Zeichenfolge ein, die der Umwandlungsanweisung entspricht, z. B. eine Ziffernfolge bei dezimalganzzahliger Umwandlung, wandelt diese um und legt das Ergebnis an der beschriebenen Stelle ab. Wenn ein Fehler auftritt, bricht `"scanf"` sofort ab.

Bei den bereits mehrfach erwähnten Umwandlungszeichen handelt es sich um die Folgenden:

Zeichen	Argument	Erwartete Eingabedaten und Umwandlung
`d`	int*	Dezimal, ganzzahlig
`i`	int*	Ganzzahlig, wahlweise dezimal, oktal (mit `"0"` beginnend) oder hex (mit `"0x"` oder `"0X"` am Anfang)
`o`	int*	Oktal, ganzzahlig
`x`	int*	Hexadezimal mit oder ohne `"0x"`
`c`	char*	So viele Zeichen, wie die Feldbreite angibt (Default ist 1), wobei Leerzeichen, Zeilentrenner o. Ä. nicht überlesen werden. Will man ein Zeichen lesen, vorher aber Zwischenraumzeichen überlesen, ist die Umwandlung `"%1s"` zu wählen. Im Gegensatz zu `"%s"` wird bei `"%c"` kein Nullbyte angehängt.
`s`	char*	Überliest zuerst Leerzeichen und Tabulatoren und liest dann eine Folge von Nicht-Zwischenraum-Zeichen. Ein Nullbyte wird an das Ergebnis angehängt.
`e,f,g`	float*	Fließkommazahl in nahezu beliebigem Format (mit oder ohne Voerzeichen, mit oder ohne Exponent, der ein großes `"E"` oder ein kleines `"e"` haben darf usw.)
`p`	int*	Hexadezimale Zahl
`n`	int*	Liest nichts, sondern legt im Argument die Anzahl der bisher gelesenen Zeichen ab.
`[...]`	char*	Längste Zeichenkette, die ausschließlich aus den Zeichen zwischen `"["` und `"]"` besteht, z. B. Ziffernfolgen bei Umwandlungsoperator `"%[0123456789]"`. Auch hier kann die Länge durch Setzen einer Feldbreite beschränkt werden. Will man das Zeichen `"]"` aufnehmen, ist es an den Anfang zu setzen (z. B. `"%[ ]abcde]"`). Die gelesenen Zeichen werden wie üblich mit einem `'\0'` abgeschlossen.
`[^...]`	char*	Liest die längste Zeichenkette, deren Zeichen sich nicht in der angegebenen Menge befinden.
`%`	-	Erwartet ein `"%"`, legt es aber nirgendwo ab.

Natürlich kann statt eines Zeigers auf `"int"` stets ebenso gut ein Zeiger auf `"unsigned int"` verwendet werden. Bei ganzzahligen Umwandlungsoperationen wird das zugehörige Zeigerargument durch die Längenangabe ‚h' als `"short"` betrachtet, `'l'` macht daraus `"long"`

und `'L'` `"long long"`. Bei den Fließkommaoperationen ist `'l'` für `"double"`-Variablen und `'L'` für `"long double"`-Variablen zu setzen.

Beispiel:

```
int i,j;
short s;
char c[80];

scanf("Test %d %i, %hi %79s", &i, &j, &s, c)
```

akzeptiert z. B. die Eingabe

```
Test 42 0x686b, 99 Rest
```

oder auch

```
Test42 26731      ,99Rest
```

Das Ergebnis von `"scanf"` ist in beiden Fällen 4, und in die Variablen werden die Werte `"42"`, `"26731"`, `"99"` und `"Rest"` eingelesen.

Mehr noch als `"printf"` ist `"scanf"` eine höchst unsichere Funktion, da keine wirkliche Typprüfung der Argumente vorgenommen wird. Wenn man „nur" ein `"&"` vergißt (und damit den Wert einer Variablen statt ihrer Adresse als Argument übergibt), ist der Hangup oft nicht mehr zu vermeiden.

fscanf

```
int fscanf(FILE *f, const char *format, ...)
```

Dies ist das Äquivalent zu `"scanf"` für beliebige Dateien.

```
scanf(arglist)
```

ist gleichbedeutend mit

```
fscanf(stdin, arglist)
```

wobei `"arglist"` hier nichts mit Niedertracht und Heimtücke zu tun hat, sondern eine Argument-Liste darstellen soll.

sscanf

```
int sscanf(char *s, const char *format, ...);
```

ist ebenfalls eine andere Form von `"scanf"`. Hier wird die Eingabe aus dem String `"s"` gelesen. Beispielsweise wandelt

```
sscanf(s, "%i", &n)
```

die Ziffernfolge `"s"` in eine Zahl um und leistet dabei ähnliches wie

```
n = atoi(s)
```

1.2.6 Dateioperationen

1.2.6.1 Dateiende und Fehlerbehandlung

feof

```
int feof(FILE *f)
```

liefert einen Wert ungleich Null, wenn in der genannten Datei das Dateiende überschritten wurde.

ferror

```
int ferror(FILE *f)
```

Diese Funktion schaut nach, ob bei der Datei `"f"` bisher ein Fehler aufgetreten ist und vermerkt wurde, und liefert die Fehlernummer oder andernfalls Null.

clearerr

```
void clearerr(FILE *f)
```

löscht die Dateiende- und Fehlervermerke einer Datei, so daß anschließend `"feof"` und `"ferror"` jeweils Null ergeben.

1.2.6.2 Puffern

setvbuf

```
int setvbuf(FILE *f, char *buf, int mode, unsigned size)
```

Normalerweise gehen Ausgabeoperationen erheblich schneller, wenn man die Daten nicht in kleinen Kleckermengen (z. B. Zeichenweise) schreibt, sondern sie erst einmal in einem Puffer sammelt und dann auf einen Schlag schreibt. Das gilt vor allem unter den alten Amiga-Systemversionen mit diesem unsäglich lahmen BCPL-DOS, aber auch unter 2.0 und höchstwahrscheinlich auch allen folgenden Versionen. Der Grund ist ganz einfach der, daß bei jeder Ausgabeoperation ein erheblicher Overhead nötig ist: Das Programm ruft eine DOS-Funktion auf, das DOS gibt an das File-System weiter, welches wiederum ein Device befragt, und erst das betätigt dann (z. B.) den Controller, der dann mit der Festplatte kommuniziert... Dieser Overhead ist für jede Datengröße praktisch identisch, so daß es natürlich Zeit spart, wenn man das ganze Prozedere nur einmal für einen großen Datenblock durchexerziert und nicht für jedes Byte einzeln. Analog gilt das natürlich auch für Eingaben aus einer Datei.

Die Funktion `"setvbuf"` versieht eine Datei mit einem solchen Puffer. Man kann sie für jede Datei nur einmal aufrufen, und zwar unmittelbar nach dem Öffnen, d. h. vor jeder Ein- und Ausgabe oder sonstigen Operation. `"buf"` zeigt dabei auf einen hinreichend großenSpeicherbereich, beim Wert Null legt `"setvbuf"` selbstständig einen solchen Puffer an und löscht ihn beim Schließen der Datei auch wieder. Der Parameter `"size"` gibt die gewünschte Puffergröße an. Erfahrungsgemäß sind hier einige hundert Bytes genug und eine weitere Steigerung bringt nicht mehr viel, und `"mode"` gibt einen der im folgenden beschriebenen Puffermodi an. Das Ergebnis von `"setvbuf"` ist Null, wenn alles klar ging.

IOFBF

```
#define _IOFBF 1
```

Beim Modus `"_IOFBF"` puffert `"setvbuf"` vollständig, d. h. der Puffer wird immer komplett gefüllt, bevor eine neue Operation erfolgt.

IOLBF

```
#define _IOLBF (-1)
```

Bei Ausgaben wird der Puffer im Modus `"_IOLBF"` höchstens bis zum ersten Zeilenende gefüllt. Dieser Modus ist z. B. auch bei Bildschirmausgaben brauchbar und bringen sogar eine gewisse Beschleunigung.

IONBF

```
#define _IONBF 0
```

Es wird nichts gepuffert. Einen Aufruf von `"setvbuf"` mit diesem Modus kann man sich also eigentlich sparen.

setbuf

```
void setbuf(FILE *f, char *buf)
```

Dies ist eine Spar-Version von `"setvbuf"`, bei der `"buf"` auf einen Puffer der Größe `"BUFSIZ"` zeigen. `"setbuf"` legt nicht selbstständig einen Speicherbereich der geforderten Größe an.

```
char b[BUFSIZ];
FILE *f;

setbuf(f, b);
```

ist äquivalent zu

```
setvbuf(f, b, _IOFBF, BUFSIZ);
```

Fehler können bei `"setbuf"` nicht auftreten, und somit ist das Ergebnis `"void"`.

BUFSIZ

```
#define BUFSIZ 200
```

ist die Puffergröße, die bei `"setbuf"` angenommen wird.

fflush

```
int fflush(FILE *f)
```

klingt zwar irgendwie gestottert, wirkt aber in Wirklichkeit einer stoßweisen Stotter-Ausgabe (will sagen: Schreiben mit Puffer) entgegen: Die Daten, die sich gerade im Puffer der Datei befinden, werden geschrieben, und der Puffer wird gelöscht. Bei Eingabe-Dateien hat `"fflush"` keine Wirkung, und wenn als Argument Null gewählt wird, werden alle gerade geöffneten Dateien geflusht.

1.2.6.3 Externe Dateien

remove

```
int remove(const char *name)
```

entfernt die Datei dieses Namens, entspricht also dem CLI-Befehl `"delete"`. Hat das geklappt, ist das Resultet Null, andernfalls die Fehlernummer.

rename

```
int rename(const char *oldname, const char *newname)
```

benennt die Datei `"oldname"` in `"newname"` um, oder versucht es zumindest. Im Erfolgsfall wird Null als Ergebnis geliefert und andernfalls eine Fehlernummer.

tmpnam

```
char *tmpnam (char s[L_tmpnam])
```

Manchmal steht man vor der Situation, daß man einige Daten in temporären Dateien ablegen will. Die Funktion `"tmpnam"` liefert dafür Dateinamen, die garantiert mit keiner existierenden Datei übereinstimmen: Sie beginnen mit `"t:TMP_"`, und im Rest der Zeichenfolge sind die Adresse der Taskstruktur, die laufende Nummer des `"tmpnam"`-Aufrufs, das Datum und die Uhrzeit kodiert.

Der Parameter `"s"` muß auf eine Stringvariable der Länge (mindestens) `"L_tmpnam"` zeigen, in der das Ergebnis abgelegt wird, oder Null sein, dann wird der Dateiname in einem internen Buffer abgelegt. In beiden Fällen wird als Ergebnis ein Zeiger auf den String geliefert. `"tmpnam"` erzeugt keine Datei, sondern schlägt nur einen Namen vor. Außerdem stehen höchstens `"TMP_MAX"` verschiedene Namen zur Verfügung, aber bei MaxonC++ sind das mehr als genug.

`L_tmpnam`

```
#define L_tmpnam 40
```

Die Konstante gibt an, wie groß ein String wenigstens sein muß, um das Ergebnis von `"tmpnam"` aufzunehmen.

`TMP_MAX`

```
#define TMP_MAX 0x10000
```

"tmpnam" kann TMP_MAX mal aufgerufen werden, bevor sie sich wiederholt. In MaxonC++ sind das offensichtlich mehr als genug mögliche Dateinamen, aber darauf sollte man sich nicht verlassen - man denke hier nur an MesSy-DOS, wo Dateinamen nur 8+3 Zeichen lang sein dürfen. Übrigens muß man sich ganz schön beeilen, wenn man mit "tmpnam" wirklich zweimal denselben Namen erhalten will, denn MaxonC++ codiert ja auch die Uhrzeit in den Namen ein.

tmpfile

```
FILE *tmpfile (void)
```

ist so etwas wie die Kombination von "tmpnam" mit "fopen" und einer Löschautomatik: Eine temporäre Datei wird mit dem Modus "wb+" geöffnet, und beim Schließen dieser Datei oder am Programmende wird sie automatisch wieder gelöscht. Wenn das Ganze ein Schlag ins Wasser war, liefert "tmpfile" Null, andernfalls einen Zeiger auf die Datei.

1.2.6.4 Binäre Daten

fread

```
unsigned fread(void *ptr,
unsigned size, unsigned n, FILE *f)
```

"fread" liest aus der Datei "f" in den Vektor "ptr" maximal "n" Objekte der Größe "size". Bitte fragen Sie nicht, warum man hier und bei "fwrite" erst "size" und dann "n" angeben soll, während es bei "calloc", "qsort" und "bsearch" genau umgekehrt ist. "fread" liefert als Ergebnis die Anzahl der gelesenen Objekte (nicht Bytes!), die natürlich keineswegs größer, aber durchaus geringer als "n" sein kann (z. B. wenn das Dateiende erreicht wurde).

fwrite

```
unsigned fwrite(const void *ptr, unsigned size, unsigned n, FILE *f)
```

ist das Gegenstück zu "fread" und schreibt "n" Elemente der Größe "size" aus dem Vektor "ptr" in die Datei "f". Resultat ist die Anzahl der geschriebenen Objekte, die im Falle eines Fehlers kleiner als "n" ist.

1.2.6.5 Positionieren in Dateien

fseek

```
int fseek(FILE *f, long offset, int modus)
```

setzt den Schreib- bzw. Lesezeiger in einer Datei an eine andere Position. Es gibt drei Modi: `"SEEK_SET"` geht an die angegebene absolute Position (Dateianfang ist Null), `"SEEK_CUR"` verschiebt den Zeiger um den Offset, und bei `"SEEK_END"` bezieht sich der Offset auf das Dateiende, also z. B. `"-1"` für das letzte Byte der Datei. Die Funktion liefert im Falle eines Fehlers einen von Null verschiedenen Wert.

```
SEEK_CUR, SEEK_SET, SEEK_END

#define SEEK_CUR 0
#define SEEK_END 1
#define SEEK_SET (-1)
```

Dies sind die drei Modi, die bei `"fseek"` angegeben werden können.

ftell

```
long ftell(FILE *f)
```

liefert die aktuelle Dateiposition von Datei `"f"`.

rewind

```
void rewind(FILE *f)
```

setzt die Datei `"f"` an den Anfang zurück und löscht das Fehlerflag. Damit ist `"rewind(f)"` eine Abkürzung für `"fseek(f, 0, SEEK_SET)"` mit anschließendem `"clearerr(f)"`.

fpos_t

```
typedef int fpos_t
```

Der Datentyp `"fpos_t"` wird gebraucht, weil einige Implementierungen zwischen Text- und Binärdateien unterscheiden und die Anwendung `"fseek"` dann bei Textdateien etwas problematisch ist. Deshalb gibt es speziell für Textdateien die beiden folgenden Funktionen:

fgetpos

```
int fgetpos(FILE *f, fpos_t *posn)
```

"fgetpos" speichert die aktuelle Position der Datei "f" in "posn", insbesondere zur späteren Verwendung mit "fsetpos". Wenn das System selbst nicht so genau weiß, was diese Posotion ist, liefert "fgetpos" einen Wert ungleich Null.

fsetpos

```
int fsetpos(FILE *f, const fpos_t *posn)
```

Das versprochene Gegenstück zu "fgetpos": Die Datei wird wieder auf die Position gesetzt, die in "posn" mittels "fgetpos" abgespeichert wurde.

1.2.7 Sonstige Funktionen

perror

```
void perror(const char *s)
```

gibt eine Fehlermeldung aus, die sich zum einen aus dem String "s" und zum anderen aus der Fehlermeldung, die zu der Nummer gehört, die gerade in "errno" steht. Hat "errno" z. B. den Wert 205, gibt

```
perror("Das war wohl nichts")
```

aus:

Das war wohl nichts: Object not found

"errno" wird eigentlich immer gesetzt, wenn irgend etwas schief geht, z. B. wenn eine Datei nicht geöffnet werden kann. Näheres steht in *<errno.h>*.

exit

```
void exit(int res)
```

lesen Sie hierzu bitte in Kapitel 1.1.5 (stdlib.h) nach.

1.3 Fehlersuche: <assert.h>

assert

```
#ifdef NDEBUG
#define assert(C)
#else
extern "Asm" void do__assert(char*, char*, unsigned int);
#define assert(C) { if(!(C)) do__assert(#C, __FILE__, __LINE__); }
#endif
```

Die Include-Datei **"assert.h"** definiert lediglich ein einziges Makro nämlich **"assert"**. Man kann damit die Sicherheit seines Programms erhöhen, indem man Plausibilitätstests einführt. Wenn z. B. eine Funktion einen Parameter zwischen 0 und 42 erwartet, setze man an ihren Anfang folgendes:

```
int f(int i)
{
assert( i>0 && i<42 )

i++;          // usw.
```

Gilt die bei **"assert"** angegebene Bedingung nicht, steigt das Programm bei einem ungültigen Parameter mit einer Meldung wie

```
Assertion failed: i>0 && i<42, file "murx.cpp", line 4711
```

aus - es sei denn, man definiert das Makro **"NDEBUG"**, bevor man es einbindet. In diesem Fall werden die **"assert"**-Anweisungen ersatzlos gestrichen und kosten so weder Laufzeit noch Speicherplatz. Man sollte **"NDEBUG"** immer erst dann definieren, wenn man der Meinung ist, das Programm sei fehlerfrei. Auf jeden Fall aber vermeide man Bedingungen mit Seiteneffekt:

```
FILE *fp;
assert(fp = fopen("Caddy.config", "r"))        // NEIN!
```

Diese Anweisung gibt zwar eine mehr oder weniger aussagekräftige Fehlermeldung ab, wenn das Programm seine Konfigurationsdatei nicht finden kann, und steigt dann aus, was durchaus sinnvoll ist - aber irgendwann kommt man vielleicht doch noch auf die Idee, **"NDEBUG"** zu definieren, und dann wird man sich wundern.

1.4 Zeichenklassen: <ctype.h>

In der Definitionsdatei *<ctype.h>* findet der Programmierer einige kleine, aber recht nützliche Funktionen, durch die Zeichen weitgehend unabhängig vom verwendeten Rechner und Zeichencode behandelt werden können. Dies ist enorm wichtig für saubere (d. h. portable) Programmentwicklung.

isalnum, isalpha, iscntrl, isdigit, isgraph, islower, isprint, ispunct, isspace, isupper, isxdigit

```
int isalnum (int c)
int isalpha (int c)
int iscntrl (int c)
int isdigit (int c)
int isgraph (int c)
int islower (int c)
int isprint (int c)
int ispunct (int c)
int isspace (int c)
int isupper (int c)
int isxdigit(int c)
```

Diese Funktionen testen, ob das Zeichen `"c"` zu einer bestimmten Zeichgenklasse gehört, und liefern dann den entsprechenden logischen Wert (also `"0"` oder `"1"`). Es gibt prinzipiell folgende Klassen von Zeichen:

- Großbuchstaben von `'A'` bis `'Z'`
- Kleinbuchstaben von `'a'` bis `'z'`
- Allgemeine Buchstaben, d. h. Groß- oder Kleinbuchstaben. Umlaute werden dabei jeweils leider nicht berücksichtigt.
- Dezimale Ziffern von `'0'` bis `'9'`
- Hexadezimale Ziffern: `'0'` bis `'9'`, `'A'` bis `'F'` sowie `'a'` bis `'f'`
- Leerraum, d. h. Leerzeichen, Zeilentrenner (Linefeed), Tabulatoren, Rücklauf (Return) und Seitenvorschub
- Sichtbare Zeichen einschließlich des Leerzeichens - das sind auf dem Amiga die Codes von `0x20` bis `0x7F` sowie von `0xA0` bis `0xFF`.
- Steuerzeichen, d. h. das genaue Gegenteil der sichtbaren Zeichen

Die folgende Liste gibt an, welche Funktion bei welcher Zeichenklasse „wahr" liefert:

isalnum Buchstabe oder Ziffer

`isalpha`	Buchstabe
`iscntrl`	Steuerzeichen
`isdigit`	Dezimale Ziffer
`isgraph`	Sichtbares Zeichen außer Leerzeichen
`islower`	Kleinbuchstabe
`isprint`	Sichtbares Zeichen
`ispunct`	Sichtbares Zeichen außer Leerzeichen, Buchstabe oder Ziffer
`isspace`	Leerraum-Zeichen
`isupper`	Großbuchstabe
`isxdigit`	Hex-Ziffer

tolower

```
int tolower(int c)
```

Die Funktion wandelt das Zeichen `"c"`, sofern es sich dabei um einen Großbuchstaben handelt, in den entsprechenden Kleinbuchstaben um. Andernfalls wird das Argument unverändert zurückgegeben. Die Funktion entspricht dabei in etwa der Fallunterscheidung

```
if (c >= 'A' && c <= 'Z')
  return c+('a'-'A');
else
  return c;
```

Auch hier werden Umlaute leider nicht berücksichtigt.

toupper

```
int toupper(int c)
```

Analog zu `"tolower"` werden hier Kleinbuchstaben in Großbuchstaben umgewandelt und andere Zeichen unverändert belassen.

which_xdigit [M]

```
int which_xdigit(char c)
```

Wenn der Parameter `"c"` eine hexadezimale Ziffer ist, wird der zugehörige numerische Wert zurückgegeben, andernfalls `"-1"`. Beispiele:

```
which_xdigit('7')  = 7
```

```
which_xdigit('b')  = 11
which_xdigit('C')  = 12
which_xdigit('+')  = -1
```

1.5 Fehlernummern: <errno.h>

Während der Laufzeit eines Programms kann so einiges schiefgehen - ein guter Programmierer geht immer davon aus, daß alles schiefgeht, was irgendwie schiefgehen kann. Die Analyse von Laufzeitfehlern wird mit den Definitionen der Datei *<errno.h>* ein wenig erleichtert.

errno

```
extern int errno
```

In dieser globalen Variablen legen viele Standard-Funktionen in Falle eines Falles eine Fehlernummer ab. Beispielsweise wird hier die entsprechende DOS-Fehlernummer zugewiesen, wenn bei `"fopen"` keine Datei geöffnet werden konnte. Vor solchen kritischen Operationen sollte man deshalb `"errno"` auf Null setzen.

ERANGE

```
#define ERANGE 1000
```

Diese Fehlernummer wird `"errno"` zugewiesen, wenn bei einer Funktion ein Wertebereich überschritten wurde.

EUSRBRK [M]

```
#define EUSRBRK 900
```

Diese Konstante gibt den Rückgabewert eines Programms an, wenn es vom Benutzer abgebrochen wurde. Obwohl die Konstante in *<errno.h>* definiert wird, wird dieser Wert dabei nicht wirklich an `"errno"` zugewiesen und kann folglich auch nicht abgefragt werden. Vielmehr wird diese Konstante definiert, damit man mit einer Anweisung wie `"exit(EUSRBRK)"` ein derartiges Programmende simulieren kann.

EASSERT [M]

```
#define EASSERT 990
```

Genau wie "EUSRBRK" ist dies kein Wert, den "errno" annimmt, sondern der Rückgabewert eines Programms, und zwar dann, wenn bei einem "assert"-Makro ein Fehler entdeckt wird.

EFREEMEM [M]

```
#define EFREEMEM 996
```

Auch dies ist ein Rückgabewert und keine wirkliche Fehlernummer. Mit diesem Wert bricht ein Programm ab, wenn bei "free" bzw. "delete" ein ungültiger Speicherbereich freigegeben werden soll.

ENONUM [M]

```
#define ENONUM 1001
```

Dieser Fehlercode wird von Funktionen wie "atoi" oder "strtod" gesetzt, wenn ein String keine gültige Zahldarstellung ist.

ENOMEM [M]

```
#define ENOMEM 1002
```

"new" und die "malloc"-Funktion setzen diese Fehlernummer, wenn kein Speicher reserviert werden konnte.

1.6 Ganzzahlige Grenzwerte: <limits.h>

Jede Implementierung kann mehr oder weniger frei festlegen, welchen Umfang die einzelnen Datentypen haben. In der Datei *<limits.h>* werden Konstanten definiert, die die spezifischen Daten eines C-Systems angeben und so standard-gemäße Programmierung erleichtern.

CHAR_BIT

```
#define CHAR_BIT 8
```

gibt die Anzahl der Bits in einem `"char"` an.

CHAR_MAX, CHAR_MIN, SCHAR_MAX, SCHAR_MIN, SHRT_MAX SHRT_MIN, INT_MAX, INT_MIN, LONG_MAX, LONG_MIN, LONGLONG_MAX, LONGLONG_MIN

```
#define CHAR_MAX        127
#define CHAR_MIN        (-128)
#define SCHAR_MAX       127
#define SCHAR_MIN       (-128)
#define SHRT_MAX        0x7fff
#define SHRT_MIN        (-0x8000)
#define INT_MAX         0x7fffffff
#define INT_MIN         (-0x80000000)
#define LONG_MAX        0x7fffffff
#define LONG_MIN        (-0x80000000)
#define LONGLONG_MAX    0x7fffffffffffffffLL
#define LONGLONG_MIN    (-0x8000000000000000LL)
```

Zu jedem vorzeichenbehafteten Datentyp existieren je zwei Konstanten, die den minimalen bzw. maximalen Wert dieses Typs angeben. Dementsprechend enden die Namen dieser Konstanten auf `"_MIN"` oder `"_MAX"` und beginnen je nach Typ mit `"CHAR"`, `"SCHAR"`, `"SHRT"`, `"INT"`, `"LONG"` oder `"LONGLONG"`.

UCHAR_MAX, USHRT_MAX, UINT_MAX, ULONG_MAX, ULONGLONG_MAX

```
#defineUCHAR_MAX 255
#defineUSHRT_MAX 65535
#defineUINT_MAX  0xffffffffU
#defineULONG_MAX 0xffffffffU
#defineULONGLONG_MAX  0xffffffffffffffffULL
```

Bei den vorzeichenbehafteten ganzzahligen Datentypen ist naturgemäß stets Null der kleinste darstellbare Wert. Deshalb existiert für diese Typen lediglich jeweils eine Konstante, die den höchsten Wert des Datentyps darstellt.

1.7 Mathematische Funktionen: <math.h>

1.7.1 Umwandlung von Zahlen in Zeichenketten

PARAMETER_BASE, PARAMETER_DIGITS [M]

```
#ifdef __cplusplus
#define PARAMETER_BASE short=10
#define PARAMETER_DIGITS short=0
#else
#define PARAMETER_BASE short
#define PARAMETER_DIGITS short
#endif
```

Die nachfolgend beschriebenen Funktionen sind allesamt Spezialitäten von MaxonC++ und wurden durch Default-Argumente etwas komfortabler gemacht. Da ANSI C aber keine Default-Argumente kennt, werden in den Parameterlisten diese beiden Makros benutzt. In C++ kann man also das jeweilige Argument weglassen, in C muß man es setzen. Übrigens werden die Makronamen am Ende von *<math.h>* mit `"#undef"` wieder gelöscht, so daß im weiteren keine Kollisionen auftreten.

inttostr, uinttostr, vlongtostr, uvlongtostr [M]

```
char *inttostr (int i, char s[ ], PARAMETER_BASE)
char *uinttostr (unsigned u, char s[ ], PARAMETER_BASE)
char *vlongtostr (long long ll, char s[ ], PARAMETER_BASE)
char *uvlongtostr (unsigned long long ull, char s[ ], PARAMETER_BASE)
```

Diese vier Funktionen wandeln ihr jeweils erstes Argument in eineZeichenkette um und legen das Ergebnis im String `"s"` ab. Der (in C++ optionale) dritte Parameter gibt die gewünschte Basis, eine Zahl von 2 bis 36, an, wobei 10 Default-Wert ist. Die umgewandelte Zeichenkette wird natürlich mit einem Nullzeichen abgeschlossen und ein Zeiger auf `"s"` zurückgegeben. Sie als Programmierer sind selbst dafür verantwortlich, daß der String `"s"` lang genug für das Ergebnis ist.

doubletostr, floattostr [M]

```
char *floattostr (float f, char s[ ], PARAMETER_DIGITS)
char *doubletostr (double d, char s[ ], PARAMETER_DIGITS)
```

Mit diesen beiden Funktionen können Sie bequem und ohne Benutzung von `"sprintf"` eine Fließkommazahl in eine Zeichenkette verwandeln. Das Ergebnis mit einem abschließenden Nullbyte wird dabei im String `"s"` abgelegt (der natürlich hinreichend lang sein sollte), und der letzte Parameter, für den im C++-Modus der Default-Wert Null definiert ist, gibt den Modus und die Ziffernanzahl an:

Bei einer positiven Anzahl von Stellen entspricht diese der Zahl der Nachkommastellen:

```
doubletostr(17.4, s, 5)
```

liefert

s = " 17.40000",

während eine negative Anzahl die Gesamtzahl der Dezimalstellen bei Exponentialdarstellung angibt:

```
doubletostr(17.4, s, -5)
```

führt zum Ergebnis

s = " 1.7400e+001".

und beim Argument `"0"` (bzw. dem Default-Argument) wird für Zahlen zwischen 10000 und 0.1 die Festpunktdarstellung mit der minimalen Anzahl von Nachkommastellen und sonst die Exponentialdarstellung gewählt.

Im Gegensatz zu den zuvor beschriebenen ganzzahligen Formatierungsfunktionen wird bei `"doubletostr"` und `"floattostr"` positiven Zahlen ein Leerzeichen vorangestellt.

1.7.2 Fließkomma-Berechnungen

sin, cos, tan, asin, acos, atan, sinh, cosh, tanh, exp, log, log10, sqrt

```
double sin (double x)
double cos (double x)
double tan (double x)
double asin (double x)
double acos (double x)
double atan (double x)
double sinh (double x)
double cosh (double x)
double tanh (double x)
double exp (double x)
double log (double x)
double log10 (double x)
double sqrt (double x)
```

Diese Funktionen stellen die wichtigsten mathematischen Funktionen dar. Offensichtlich sind ihr Argument und Ergebnis jeweils `"double"`-Werte. Die Funktionen haben folgende (meist offensichtliche) Bedeutung:

`sin`	Sinus
`cos`	Cosinus
`tan`	Tangens
`asin`	sin ^ -1 für Argumente zwischen -1 und 1
`acos`	cos ^ -1 für Argumente zwischen -1 und 1
`atan`	tan ^ -1, Ergebnis zwischen -pi/2 und +pi/2
`sinh`	Sinus Hyperbolicus
`cosh`	Cosinus Hyperbolicus
`tanh`	Tangens Hyperbolicus
`exp`	Exponentialfunktion e ^ x
`log`	Natürlicher Logarithmus
`log10`	Logarithmus zur Basis 10
`sqrt`	Quadratwurzel

Bei den trigonometrichen Funktionen werden Winkel natürlich im Bogenmaß (Radian) angegeben. Ein Gradwinkel `"w"` läßt sich mit der Formel

```
x = 3.14159265358979/180.0 * w
```

in Bogenmaß umrechnen.

pow

```
double pow (double x, double y)
```

Diese Funktion berechnet die Potenz x ^ y, wobei aber `"x"` positiv sein muß.

atan2

```
atan2(double x, double y)
```

berechnet den Arcustangens von `y/x`, d. h. den Winkel, der zum Cosinus `"x"` und dem Sinus `"y"` gehört. Das Ergebnis ist aus `[-pi, +pi]`, und die Funktion ist besonders dann praktisch, wenn man kartesische in Polarkoordinaten umrechnen will.

floor

```
floor(double x)
```

Die Funktion `"floor"` rundet auf die nächstkleinere ganze Zahl ab (genaugenommen die größte ganze Zahl kleiner oder gleich x). Das Ergebnis ist nichtsdestotrotz ein `"double"`- und kein `"int"`-Wert.

ceil

```
double ceil(double x)
```

Analog zu `"floor"` rundet `"ceil"` ihr Argument auf.

fabs

```
fabs(double x)
```

`"fabs"` liefert den Betrag ihres Arguments `"x"`.

ldexp

```
ldexp(double x,int n)
```

Mit dieser Funktion erzeugt man aus einer Mantisse `"x"` und einem Binär-Exponenten `"n"` eine Fließkommazahl `"x * 2^n"`. `"ldexp"` ist vor allem dann nützlich und auch ausgesprochen schnell, wenn man einen `"double"`-Wert mit einer Zweierpotenz multiplizieren will.

frexp

```
double frexp(double x ,int *expo)
```

`"frexp"` ist das Gegenstück zu `"ldexp"`, denn hier wird das Argument in eine normalisierte Mantisse im Bereich von 0.5 bis 1 und einen Exponenten zur Basis 2 zerlegt. Die Mantisse wird als Funktionswert zurückgegeben, während der Exponent in `"*expo"` abgelegt wird.

modf

```
double modf(double x, double *dp)
```

Die Zahl `"x"` wird in ihre Vor- und Nachkommastellen aufgeteilt. Die Nachkommastellen, also eine Zahl zwischen -1 und +1, sind das Resultat der Funktion, während der ganzzahlige Teil nach `"*dp"` geschrieben wird. Beide Ergebnisse haben das gleiche Vorzeichen wie `"x"`.

fmod

```
double fmod(double x, double y)
```

Diese Funktion sollte nicht mit `"modf"` verwechselt werden, sie berechnet den Rest, der bei einer Fließkommadivision `"x/y"` bei ganzzahligem Quotienten auftritt.

pwr10 [M]

```
double pwr10(int i)
```

Für ganzzahlige `"i"` im Bereich von `-308` bis `+308` liefert `"prw10"` die Potenz `10^i`, und zwar ziemlich schnell, denn die Werte werden dabei teilweise einer Tabelle entnommen.

expo10 [M]

```
int expo10(double x)
```

Diese Funktion liefert die größte ganze Zahl "n", für die "prw10(n)" nicht größer als "x" ist, also den Exponententeil von "x". Nicht ganz zufällig sind "pwr10" und "expo10" Abfallprodukte der Stringkonvertierungsfunktionen von Fließkommazahlen.

fpwr10 [M]

```
float fpwr10(int i)
```

"fpwr10" ist das "float"-Gegenstück zu "pwr10" und berechnet ebenfalls die Potenz "10^i", aber als "float"-Wert.

1.8 Haarsträubende Sprünge: <setjmp.h>

setjmp, longjmp

```
int setjmp(jmp_buf buf)
void longjmp(jmp_buf buf, int num)
```

Die normalen "goto"-Sprünge können nur innerhalb einer Funktion herumspringen, und das ist oft nicht genug. Manchmal muß man von einer Funktion in eine andere zurückspringen, z. B. wenn irgendwo in einer tief verschachtelten Reihe von Funktionsaufrufen ein Fehler auftritt und die Programmfunktion, aber nicht das ganze Programm abgebrochen werden soll. Es wäre dabei meist sehr umständlich, hinter jeden der zahllosen Funktionsaufrufe eine Abfrage der Art „Ist-ein-Fehler- aufgetreten-wenn-ja-dann-Ende" zu setzen. Deshalb gibt es die beiden Funktionen "setjmp" und "longjmp", mit denen man direkt aus einer Funktion in eine andere, höher gelegene zurückspringen kann.

Man kann dabei keineswegs wild in irgendeine Funktion hineinspringen, denn dann wäre der Absturz nahezu garantiert. Vielmehr speichert "setjmp" die wichtigsten Daten eines Zustands, nämlich den Stackpointer, den Programmzähler und die übrigen Prozessorregister, in einem Puffer des Typs "jmp_buf" ab, und "longjmp" stellt eben diesen Zustand wieder her, aber nur dann, wenn die Funktion, in der "setjmp" aufgerufen wurde, bisher nicht beendet wurde.

Der Ablauf ist also immer der folgende: In einer Funktion "f1" wird "setjmp" aufgerufen, und damit merkt sich das Programm haargenau die Position dieses "setjmp"-Aufrufs. Nun kann aus "f1" eine andere Funktion "f2" aufgerufen werden, welche möglicherweise wiederum irgendwann eine Funktion "f3" aufruft, in der ein Aufruf von "f4" stattfindet... Jedenfalls

befindet das Programm sich irgendwann in einer Funktion `"fx"`, und dann wird mit `"longjmp"` der abgespeicherte Zustand wiederhergestellt: Die Funktion `"fx"` wird beendet und alle ihre Daten sowie die der anderen Funktionen bis einschließlich `"f2"` werden vom Stack entfernt. Die Programmausführung wird in `"f1"` an genau der Stelle, an der das `"setjmp"` steht, fortgesetzt.

Nach jedem `"setjmp"` sollte das Programm dann feststellen, was passiert ist: Wurde soeben `"setjmp"` ausgeführt und der Programmzustand abgespeichert, oder ist ein `"longjmp"` erfolgt? Im letzteren Fall befindet sich das Programm schließlich scheinbar auch direkt hinter dem Aufruf von `"setjmp"`. Deshalb hat `"setjmp"` einen Ergebniswert, der im Prinzip immer Null ist, und der Funktion `"longjmp"` kann ein numerisches Argument übergeben werden. Beim `"longjmp"`-Sprung wird dem Programm dann vorgegaukelt, `"setjmp"` habe dieses Argument als Ergebnis geliefert.

Am besten wird die Sache wohl an einem Beispiel deutlich. Eine typische Anwendung von `"setjmp"` und `"longjmp"` sieht so aus:

```
#include <setjmp.h>

jmp_buf Ausstieg;

void f2()
{
  printf("Funktion f2:\n");

  longjmp(Ausstieg,17);

  printf("PANIC\n");     // Diese Stelle wird nie erreicht.
}

void f1()
{
  if(setjmp(Ausstieg) == 0)
  // Ergebnis 0: Es wurde wirklich bloß der Zustand gespeichert
  {
    printf("Jump gesetzt.\n");
    f2();
    // Diese Stelle wird nur erreicht,
    // wenn kein "longjmp" gemacht wird:
    printf("DOUBLE PANIC\n");
  }
  else
    // Ergebnis 17: longjmp wurde ausgeführt
    printf("Jump ausgeführt.\n");
}
```

Die Funktion `"f1"` macht dabei folgende Ausgabe:

Jump gesetzt.

Funktion f2:

Jump ausgeführt.

`"jmp_buf"` ist stets ein Vektortyp, weshalb bei `"setjmp"` kein Adressoperator `"&"` vor das Argument gesetzt werden muß. Ihnen dürfte (oder sollte) klar sein, daß diese Funktion in der Regel ziemlich hanebüchen implementiert wird. Deshalb darf man `"setjmp"` nur in ganz bestimmten Zusammenhängen benutzen:

- Man darf auf `"setjmp"` unäre Operatoren, z. B. `"-"` oder `"!"`, anwenden.
- Das Ergebnis kann mit den Operatoren `"=="`, `"!="`, `"<"`, `">"`, `"<="` oder `">="` mit einem konstanten numerischen Ausdruck verglichen werden.
- Der Ausdruck kann „einfach so" als Anweisung stehen oder bei `"if"`, `"while"`, `"do"` oder `"switch"` als Bedingung verwendet werden.

Ein Ausdruck wie

```
printf("Ergebnis: %d\n", setjmp(x))
```

würde also unter den meisten Implementierungen (einschließlich MaxonC++) beim anschließenden `"longjmp"` zu einem gnadenlosen Absturz führen. Selbst ein scheinbar so harmloses Konstrukt wie

```
int i;
i = setjmp(x);
if (i == 0)
{ usw.
```

ist nicht zulässig (auch wenn es bei MaxonC++ zufällig klappt).

Es gibt noch einige andere Einschränkungen: Da `"longjmp"` ein echtes Low-Level-Konstrukt ist, das den Compiler praktisch heimtückisch überlistet, werden zwar die automatischen Variablen der solchermaßen „abgewürgten" Funktionen vom Stack entfernt, aber keine Destruktoren aufgerufen. Nach der Rückkehr durch `"longjmp"` müssen Sie auch damit rechnen, daß die Optimierungen des Compilers Ärger machen und automatische Variablen jener Funktion, die nicht als `"volatile"` deklariert sind, undefinierte Werte haben.

jmp_buf

```
typedef int jmp_buf[_JMP_BUF_SIZE]
```

Dieser Datentyp repräsentiert einen geeigneten Puffer, in dem `"setjmp"` einen Programmzustand abspeichern kann.

JMP_BUF_SIZE [M]

```
#define _JMP_BUF_SIZE 16
```

Eine kleine, unbedeutende Hilfskonstante, die die Größe des Vektortyps `"jmp_buf"` angibt.

1.9 Signale und Ereignisse: <signal.h>

signal

```
void (*signal(int sig, void(*f)(int)))(int)
```

oder besser:

```
typedef void (*P2F)(int)
P2F signal (int sig, P2F f)
```

Die Funktion `"signal"` setzt eine Funktion, die aufgerufen werden soll, wenn ein bestimmtes Signal `"sig"` eintrifft. In der vorliegenden Version von MaxonC++ ist die Funktion `"signal"` weitgehend Makulatur, denn bisher wird ein Signal ausschließlich vom Programm selbst durch die Funktion `"raise"` gesendet. Der Parameter `"sig"` gibt die jeweilige Signalnummer - in der Regel eines der im folgenden beschriebenen Makros - an, und `"f"` ist ein Zeiger auf eine Funktion, die ein `"int"` als Argument erhält und kein Ergebnis (also `"void"`) liefert. Das Ergebnis von `"signal"` ist ein Zeiger auf die Funktion, die vorher als Signal-Handler gesetzt war.

Anschließend wird die Funktion `"f"` stets beim Eintreffen eines Signals `"sig"` aufgerufen und erhält dabei als Argument die Signalnummer, so daß eine Funktion durchaus als Handler für mehrere Signale verwendet werden kann.

Anstelle eines Zeigers auf eine Funktion kann auch eins der unten beschriebenen Makros `"SIG_IGN"`, `"SIG_DFL"` oder `"SIG_ERR"` verwendet werden.

raise

```
int raise(int sig)
```

Die Funktion `"raise"` sendet das Signal `"sig"` und löst so den Aufruf des entsprechenden Handlers aus.

SIGTERM, SIGABRT, SIGPFE, SIGILL, SIGINT, SIGSEGV

```
#defineSIGTERM0  // Normales Programmende
#defineSIGABRT1  // Anormales Programmende wie bei "abort"
#defineSIGFPE 2  // Arithmetikfehler
#defineSIGILL 3  // Illegale Prozessor-Anweisung
#defineSIGINT 4  // Software-Interrupt
#defineSIGSEGV5  // Segmentverletzung bei Speicherzugriff
```

Dies sind die sechs Standard-Signalnummern und damit die einzigen wirklich legalen Argumente für `"signal"` und `"raise"`. Eine Implementierung darf aber auch noch zusätzliche eigene Signale haben.

SIG_IGN

```
#define SIG_IGN ((void(*)(int))0)
```

Wird bei `"signal"` als Handlerfunktion `"SIG_IGN"` angegeben, so wird das entsprechede Signal in Zukunft ignoriert.

SIG_DFL

```
#define SIG_DFL ((void(*)(int))1)
```

Diese Pseudo-Handlerfunktion stellt als Argument bei `"signal"` den jeweiligen Default-Handler dar.

SIG_ERR

```
#define SIG_ERR ((void(*)(int))-1)
```

Die Funktion `"signal"` liefert das Ergebnis `"SIG_ERR"`, wenn irgendein Fehler aufgetreten ist, z. B. die Argumente illegal waren.

sig_atomic_t

```
typedef int sig_atomic_t
```

Für echte Signal-Freaks gibt es den ganzzahligen Datentyp **"sig_atomic_t"**, der zur Kommunikation zwischen Programm und Handler benutzt werden kann. Die „besondere" Eigenschaft dieses Datentyps ist, daß ein Zugriff auf eine solche Variable eine unteilbare Operation ist, d. h. nicht unterbrochen werden kann. Wenn man einer Variablen dieses Typs einen Wert zuweist, kann man darauf vertrauen, daß ein währenddessen eintreffendes Signal entweder vor oder nach dieser Operation ausgeführt wird und nicht etwa mittendrin, wenn die Variable so etwas wie einen Zwischenwert hat.

Oder drücken wir es einmal so aus: Eine **"long long"**-Variable besteht unter MaxonC++ aus zwei Langworten, die bei einer Wertzuweisung nacheinander geschrieben werden. Hier könnte es durchaus vorkommen, daß ein Signal-Handler genau dann aufgerufen wird, wenn erst eines der beiden Langworte geschrieben ist. Der Wert der Variablen wäre dann während der Ausführung des Handlers irgendwie eine undefinierte Mischung aus dem alten und dem neuen Wert. Wenn man also in einer globalen Variablen Parameter an einen Handler übergeben möchte, sollte man dafür ausschließlich den Datentyp **"sig_atomic_t"** benutzen.

1.10 Variable Argumentlisten: <stdarg.h>

va_list

```
typedef unsigned int va_list
```

Funktionen wie **"printf"** und **"scanf"** haben eine Parameterliste, in der jeweils nur die ersten Parameter einen Namen und Typ haben und anschließend beliebig viele weitere Argumente erlaubt sind. Syntaktisch wird dies in C++ durch die Ellipse **"..."** am Ende der Parameterliste deklariert, und man kann dieses Feature auch in selbstgeschriebenen Funktionen verwenden. Nun wird sich der geneigte Leser vielleicht fragen, wie eine Funktion denn solche namenlose Parameter mit nahezu beliebiger Anzahl und unterschiedlichsten Typen auswerten kann. Genau dabei hilft die Includedatei „*<stdarg.h>*", die einige Makros deklariert, mit denen man Stack-Argumente gleichermaßen komfortabel und portabel auswerten kann.

Das Prinzip beruht darauf, daß ein Zeiger hinter den letzten normal deklarierten Parameter gesetzt wird und dann der Reihe nach die restliche Argumentliste durchläuft. Dieser „Zeiger" wird repräsentiert durch eine Variable des Typs **"va_list"**.

va_start

```
#define va_start(AP,LASTARG) (AP)=(unsigned int)(&LASTARG) \
+sizeof(LASTARG)
```

Ein Argumentlisten-Zeiger des Typs `"va_list"` wird normalerweise initialisiert, indem er auf die erste Adresse hinter dem letzten Parameter gesetzt wird, also auf genau die Speicheradresse, an der die formal unbekannten, restlichen Argumente anfangen. Dazu dient das Makro `"va_start"`, z. B. so:

```
void fun(const char *format...)
{
  va_list v;
  va_start(v, format);

  // usw.
}
```

va_arg

```
#define va_arg(AP,TYPE) ((AP)+=sizeof(TYPE), \
sizeof(TYPE)>1 ? ((AP)=(AP)+1&(-2)):(AP), \
((TYPE*)AP)[-1])
```

Das Makro `"va_arg"` mit seiner ziemlich wüsten Definition ist in seiner Anwendung um so einfacher: Ein Aufruf der Form

```
x = va_arg(vp, TYPE);
```

liest das nächste Argument des Typs `"TYPE"` aus der variablen Argumentliste, liefert es als Ergebnis und setzt nebenbei den Argumentzeiger `"vp"` entsprechend weiter. Wenn man also die auftretenden Datentypen kennt (bei `"printf"` gibt man diese ja im Formatstring an), kann man mit einer Folge von `"va_arg"`-Aufrufen nacheinander die Argumente lesen.

Beachten Sie aber, daß „kurze" ganzzahlige Typen (short, char) bei dieser Art der Wertübergabe automatisch nach `"int"` sowie `"float"` nach `"double"` umgewandelt werden. Deshalb ist z. B. `"va_arg(x, unsigned short)"` generell unsinnig und ist durch `"va_arg(x, unsigned int)"` zu ersetzen.

va_end

```
#define va_end(AP)
```

Bei einigen Implementierungen kann die Auswertung variabler Argumentlisten ein ziemicher Akt sein, und deshalb gibt es das Makro `"va_end"`, mit dem man immer eine Argumentauswertung beenden sollte und das bei Bedarf irgendwelche Aufräumarbeiten durchführt. Bei MaxonC++ ist `"va_end"` ein leeres Makro, aber trotzdem sollte man es nie vergessen.

Zum Abschluß ein zusammenhängendes Beispiel für die Auswertung variabler Argumentlisten. Der Einfachheit halber ist die Argumentliste hier überhaupt nicht variabel, sondern es wird stets erwartet, daß ein `"int"`-, ein `"double"`- und ein `"char"`-Argument folgen. Trotzdem sollte das Prinzip klar werden:

```
#include <stdarg.h>
#include <stdio.h>

void fun(const char *format...)
{
  va_list v;
  int i;
  double d;
  char c;

  va_start(v, format);  // initialisieren mit letztem bekannten
  // Parameter

  // die drei Argumente werden der Reihe nach gele     sen:
  i = va_arg(v, int);
  d = va_arg(v, double);
  c = (char) va_arg(v, int); // Auch chars werden als "int"
        // Übergeben

  // der Ordnung halber aufräumen:
  va_end(v);

  printf("%s: %i , %g , %c\n", format, i, d, c);
}

void main()
{
  fun("Test", 42, 47.11, 'x');
}
```

1.11 Implementationsabhängige Definitionen: <stddef.h>

NULL

```
#define NULL 0
```

Die allgemein beliebte und geschätzte Konstante `"NULL"` wird auch in „*<stddef.h>*“ deklariert.

offsetof

```
#define offsetof(s,m) ((unsigned)&((s*)NULL)->m)
```

In ANSI C gibt es keine Pointer auf Member. Als die C-Programmierer daraufhin neidisch nach C++ schielten, kamen sie offensichtlich auf die Idee, dieses Feature in C zu simulieren, indem sie den Offset eines Members innerhalb einer Struktur berechnen. Mit dem Makro `"offsetof"` wird dies erheblich erleichtert: `"offsetof(s,m)"` liefert den Offset (in Bytes) des Members `"m"` in der Klasse oder Struktur `"s"`.

size_t

```
typedef unsigned size_t
```

`"size_t"` wird nicht nur in „*<stdlib.h>*“ und „*<stdio.h>*“, sondern auch in `"stddef.h"` definiert und stellt den Typen dar, den das Ergebnis von `"sizeof"` in einer bestimmten Implementation hat.

ptrdiff_t

```
typedef int ptrdiff_t
```

Subtrahiert man zwei Zeiger gleichen Typs voneinander, so ist das Ergebnis bekanntlich ein ganzzahliger Wert, der die Anzahl der Vektorelemente zwischen diesen Zeigern angibt. Nun kann der Typ dieser Differenz durchaus implementationsabhängig sein: Bei einigen Compilern muß eine Adressdistanz als `"long int"` ausgedrückt werden, bei anderen (wie MaxonC++) reicht auch ein `"int"`. Deshalb gibt es für diesen Typ den standardisierten Namen `"ptrdiff_t"`.

wchar_t

```
typedef int wchar_t
```

Noch eine Wiederholung: Der Name `"wchar_t"` bezeichnet den Datentyp von langen Zeichenkonstanten.

1.12 Zeichenketten und Speicherverwaltung: <string.h>

strcpy

```
char *strcpy (char *s1, const char *s2)
```

Das Vektorkonzept von ANSI C ist bekanntlich „far beyond repair" und hat unter anderem zur Folge, daß man Strings nicht mit einem einfachen `"="` zuweisen kann. Deshalb gibt es die überaus nützliche Funktion `"strcpy"`, die einen String `"s2"`, der mit einem Nullbyte enden sollte, in eine Stringvariable `"s1"` kopiert. Das Ergebnis ist undefiniert, wenn die beiden Strings sich überlappen, und wie eigentlich immer haben Sie höchstpersönlich darüber zu wachen, daß die Zielvariable für den String `"s2"` lang genug ist. Das Ergebnis von `"strcpy"` ist der Zeiger `"s1"`.

strncpy

```
char *strncpy(char *s1, const char *s1, size_t n)
```

Die sicherheitsbetonte Variante von `"strcpy"` kopiert maximal `"n"` Zeichen aus `"s2"` nach `"s1"` und füllt den Rest mit Nullzeichen auf.

strcat

```
char *strcat (char *s1, const char *s2)
```

In diesem Fall heißt `"cat"` nicht „Katze", sondern ist eine Verbalhornung von `"concat"` oder auf Deutsch **„konkatinieren"** - man kann auch „aneinanderhängen" dazu sagen. Die Funktion sucht sich das Ende des Zeichenvektors `"s1"` (am obligatorischen Nullbyte unschwer zu erkennen) und kopiert den String `"s2"` genau dorthin, wodurch die beiden Strings zusammengehängt werden. Auch hier ist es der Sorgfalt und Umsicht des Programmierers anheim gestellt, dafür zu sorgen, daß im String `"s1"` auch noch genug Platz ist. Der Rückgabewert ist wieder einmal ein Zeiger auf das Ergebnis, also `"s1"`.

strncat

```
char *strncat(char *s1, const char *s2, size_t n)
```

Genau wie bei `"strcat"` wird auch hier der String `"s2"` an den Inhalt des Zeichenvektors `"s1"` angehänht, aber hier werden maximal `"n"` Zeichen kopiert.

strcmp

```
int strcmp (const char *s1, const char *s2)
```

Diese Funktion vergleicht die beiden Zeichenketten `"s1"` und `"s2"` und liefert:

- einen negativen Wert, wenn `s1 < s2`,
- Null, falls `s1 == s2`,
- eine positives Ergebnis, sofern `s1 > s2`.

Natürlich werden hier wirklich die Zeichen des Strings verglichen und nicht etwa die Adressen, wie es ja bei gewöhnlichen Stringvergleichen der Fall ist.

strncmp

```
int strncmp(const char *s1, const char *s2, size_t n)
```

`"strncmp"` entspricht `"strcmp"`, vergleicht aber maximal die ersten `"n"` Zeichen, oder auch weniger, wenn eher auf ein Nullzeichen und damit das Stringende gestoßen wird.

stricmp [M]

```
int stricmp(const char *s1, const char *s2)
```

Diese Funktion vergleicht die Zeichenketten `"s1"` und `"s2"` genau wie `"strcmp"`, wandelt aber zuerst alle Klein- in Großbuchstaben um, bevor zwei Zeichen verglichen werden. Diese Umwandlung geschieht natürlich nur intern, so daß es keinen Seiteneffekt auf die Strings `"s1"` und `"s2"` gibt. Beispielsweise liefert `"stricmp("Test", "TEST")"` Null.

strlwr [M]

```
char *strlwr(char *s)
```

In der durch ein Nullzeichen abgeschlossenen Zeichenkette `"s"` werden alle Großbuchstaben durch Kleinbuchstaben ersetzt. Ein Zeiger auf den solchermaßen veränderten String wird zurückgegeben.

strupr [M]

```
char *strupr(char *s)
```

funktioniert wie `"strlwr"`, wandelt aber Klein- in Großbuschtaben um.

strchr

```
char *strchr (const char *s, int c)
```

Die Funktion sucht das erste Zeichen `"c"` im String `"s"` und liefert einen Zeiger darauf. Sollte in `"s"` kein `"c"` vorkommen, ist das Ergebnis Null.

strrchr

```
char *strrchr(const char *s, int c)
```

leifert analog zu `"strchr"` einen Zeiger auf das letzte `"c"` in `"s"` oder Null, falls es so etwas nicht gibt.

strspn

```
size_t strspn (const char *s, const char *c)
```

liefert die Anzahl der Zeichen am Anfang von `"s"`, die alle auch in `"c"` vorkommen, z. B.

```
strspn("caddy", "abcd") == 4
```

strcspn

```
size_t strcspn(const char *s, const char *c)
```

ermittelt die Anzahl der Zeichen am Anfang von `"s"`, die allesamt nicht im String `"c"` vorkommen.

strpbrk

```
char *strpbrk(const char *s, const char *c)
```

ist eine Art erweiterte Version von `"strchr"` und liefert einen Zeiger auf die erste Stelle in `"s"`, an der irgendein Zeichen aus `"c"` vorkommt.

strstr

```
char *strstr(const char *s1, const char *s2)
```

Diese Funktion sucht die komplette Zeichenkette `"s2"` im String `"s1"` und liefert in gewohnter Manier einen Zeiger auf ihr erstes Auftreten oder Null.

strlen

```
size_t strlen(const char *s)
```

liefert die Länge des Strings `"s"`, wobei das Nullzeichen am Ende nicht mitgerechnet wird.

strerror

```
char *strerror(int n)
```

liefert die Fehlermeldung zur Fehlernummer `"n"`, also im Wesentlichen das, was `"perror"` ausgibt.

strtok

```
char *strtok(char *s, const char *c)
```

Mit der Funktion "strtok" kann man eine Zeichenkette "s" in eine Folge von Zeichenketten, die jeweils durch Zeichen aus "c" begrenzt sind, zerlegen. Beim ersten Aufruf ist für "s" ein Zeiger auf den Anfang der Zeichenkette zu setzen, bei allen weiteren ist hier "Null" zu setzen. Die Funktion sucht jeweils die erste Position im String, an der ein Zeichen aus "c" auftritt, setzt an diese Stelle ein '\0' und gibt einen Zeiger auf den Anfang dieses Teilstrings zurück. Dabei merkt sie sich die Position, an der das Zeichen aus "c" gefunden und das Nullzeichen gesetzt wurde, und sucht beim nächsten Aufruf, sofern als erstes Argument Null übergeben wird, von dieser Stelle weiter.

Wenn auf diese Weise der gesamte String verhackstückt wurde und kein Zeichen aus "c" mehr gefunden wurde, liefert "strtok" das Ergebnis Null.

Der sittliche Nährwert dieser Funktion liegt darin, eine Zeichenfolge "s" in eine Folge von Token, die durch Trennzeichen aus "c" separiert werden, zu zerbröseln. Beispielsweise zerlegt das folgende Programm einen String in Worte, wobei eine Zeichenfolge, die keine Leerzeichen oder Zeilentrenner enthält, als Wort betrachtet wird:

```
#include <string.h>
#include <stdio.h>

void main()
{
  char s[ ] = "Alles Seiende entsteht ohne Grund,\nsetzt sich aus
  Schwäche fort und stirbt durch Zufall.", trenn[ ] = " \n";
  char *pos;
  int i = 0;

  // erstes Wort abtrennen:
  pos = strtok(s,trenn);

  while(pos) // d. h, solange noch Worte gefunden werden:
  {
    // Wort ausgeben:
    printf("Wort %2i: %s\n", ++i, pos);

    // nächstes Wort ermitteln:
    pos = strtok(0,trenn);
  }
}
```

Das Programm liefert folgende Ausgabe:

```
Wort 1: Alles
Wort 2: Seiende
Wort 3: entsteht
Wort 4: ohne
Wort 5: Grund,
Wort 6: setzt
Wort 7: sich
Wort 8: aus
Wort 9: Schwäche
Wort 10: fort
Wort 11: und
Wort 12: stirbt
Wort 13: durch
Wort 14: Zufall.
```

memcpy

```
void *memcpy(void *v1, const void *v2, size_t n)
```

Von Adresse `"v2"` werden `"n"` Bytes an Adresse `"v1"` kopiert. Dabei dürfen sich die beiden Speicherbereiche nicht überlappen. Ergebnis der Funktion ist `"v1"`.

memmove

```
void *memmove(void *v1, const void *v2, size_t n)
```

entspricht `"memcpy"`, aber hier dürfen sich der Quell- und der Zielbereich auch überlappen.

memcmp

```
int memcmp(const void *v1, const void *v2, size_t n)
```

An den Adressen "v1" und "v2" werden die jeweils ersten "n" Zeichen miteinander verglichen. Diese Funktion ist identisch mit "strncpy", aber sie vergleicht beliebige Daten (allerdings immer byteweise) und bricht nicht beim ersten Nullbyte ab.

memchr

```
void *memchr(const void *v, int c, size_t n)
```

sucht analog zu "strchr" das Zeichen "c" in den ersten "n" Zeichen des Speichervektor "v" und liefert einen Zeiger auf dieses Zeichen.

memset

```
void *memset(void *v, int c, size_t n)
```

schreibt das Zeichen "c" in die ersten "n" Bytes ab Adresse "v".

1.13 Datum und Uhrzeit: <time.h>

clock_t

```
typedef unsigned clock_t
```

Der ganzzahlige Datentyp "clock_t" repräsentiert eine Zeitangabe und wird im Zusammenhang mit der Funktion "clock" benutzt.

time_t

```
typedef unsigned time_t
```

Auch "time_t" ist ein Datentyp und stellt so etwas wie eine komprimierte Form der "DateStamp"-Struktur des Betriebssystems dar.

tm

```
struct tm
{ int tm_sec,  tm_min,  tm_hour, tm_mday, tm_mon, tm_year,
     tm_wday, tm_yday, tm_idst; }
```

Die Struktur "tm" enthält alles, was zu einer Zeit- und Datumsangabe gehört. Die einzelnen Felder bedeuten dabei:

tm_sec	Sekunden (0 bis 61, denn man hat hier lustigerweise sogar daran gedacht, daß es bis zu 2 Schaltsekunden geben kann)
tm_min	Minuten (0 bis 59)
tm_hour	Volle Stunden seit Mitternacht (0 bis 23)
tm_mday	Monatstag (1 bis 31)
tm_mon	Monate seit Januar (d. h. von 0 bis 11!)
tm_year	Jahre seit 1900, z. B1992
tm_wday	Wochentag: von 0 (Sonntag) bis 6 (Samstag)
tm_yday	Tage seit dem 1. Januar (0 bis 365)
tm_idst	positiv, wenn Sommerzeit gilt, Null bei Normalzeit und negativ, wenn das unbekannt ist (in MaxonC^{++} immer)

time

```
time_t time (time_t *tp)
```

Die Funktion "time" liefert die aktuelle Systemzeit in Form einer "time_t"-Zahl. Wenn ihr Argument nicht Null ist, wird das Ergebnis zusätzlich noch in "*tp" abgelegt.

Normalerweise benutzt man das Ergebnis von "time" mit den Funktionen "difftime", "gmtime" oder "localtime".

gmtime

```
struct tm *gmtime (const time_t *tp)
```

Der Inhalt von "*tp", also eine Zeitangabe in diesem geheimnisvollen "time_t"-Format, wird in einem ebenso geheimen, internen Puffer in das anwenderfreundliche "tm"-Format umgewandelt. Ein Zeiger auf diesen Puffer wird zurückgegeben, so daß man die Kalender- und Zeitdaten von dort auslesen kann. Es gibt aber nur einen internen Puffer, so daß die Daten beim

nächsten Aufruf von "gmtime" oder ihrer Schwesterfunktion "localtime" wieder überschrieben werden.

localtime

```
struct tm *localtime(const time_t *tp)
```

Diese Funktion unterscheidet sich von "gmtime" nur darin, daß die Systemzeit in "*tp" hier in Ortszeit umgewandelt wird (daher das "local"), während „gmtime" ein Ergebnis in „Coordinated Universal Time" (UTC) liefern sollte - das "gm" leitet sich übrigens von der alten Bezeichnung „Greenwich Mean Time" ab. Unter MaxonC++ sind die beiden Funktionen aber absolut identisch.

mktime

```
time_t mktime (struct tm *tp)
```

"mktime" ist das Gegenstück zu "localtime" und wandelt den Inhalt der "tm"-Struktur "*tp" in das komprimierte "time_t"-Format um. Im Prinzip wird dabei verlangt, daß alle, aber auch wirklich alle Einträge der Struktur korrekt initialisiert sind. MaxonC++ ist aber schon zufrieden, wenn die ersten sechs Member (Sekunden, Minuten, Stunden, Tag, Monat, Jahr) mit gültigen Werten initialisiert sind.

clock

```
clock_t clock (void)
```

Diese Funktion liefert die bisherige Laufzeit des Programms. MaxonC++ implementiert diese Funktion etwas nachlässig, denn "clock" liefert nur die Zeit, die seit dem Programmstart vergangen ist, und nicht die CPU-Zeit, wie es eigentlich verlangt wird. Jedenfalls können Sie das Ergebnis von "clock" in Sekunden umrechnen, indem Sie durch "CLOCKS_PER_SEC" dividieren.

CLOCKS_PER_SEC

```
#define CLOCKS_PER_SEC 50
```

Diese Konstante gibt den Umrechnungsfaktor zwischen "clock_t" und einer Zeitangabe in Sekunden an. "clock()/CLOCKS_PER_SEC" liefert die bisherige Laufzeit des Programms in Sekunden.

difftime

```
double difftime(time_t t1, time_t t2)
```

"difftime" subtrahiert die Zeiten "t1-t2" und liefert das Ergebnis als Fließkommazahl in Sekunden.

strftime

```
int strftime(char *s, unsigned len, const char *format, const struct
tm *t)
```

Diese hochkomfortable Funktion wandelt die Zeit in der Struktur "*t"ähnlich wie die Funktion "sprintf" in eine Stringdarstellung um und legt diese mit einem abschließenden Nullzeichen im Zeichenvektor "s" ab. Dabei werden maximal "len" Zeichen in "s" geschrieben.

"format" ist ein Formatstring, der wie bei "sprintf" nach "s" kopiert wird. Dabei werden Anweisungsfolgen, die jeweils mit einem "%" beginnen, durch ansprechend formatierte aus "*t" ersetzt. Der Rückgabewert ist die Anzahl der Zeichen, die in "s" geschrieben wurden, wobei das abschließende Nullbyte nicht mitgezählt wird. Wenn aber "len" Zeichen nicht reichen, ist das Ergebnis Null.

Die folgende Tabelle gibt die zahlreichen möglichen Umwandlungsoperatoren an:

"	Gekürzter Wochentag, z. B. "Mon" für Montag
%A	Vollständiger Wochentag, z. B. "Monday"
%b	Monats-Kurzname, z. B. "Apr" füt April
%B	Voller Name des Monats, z. B. "April"
%c	Kurzdarstellung von Datum und Uhrzeit, z. B. "Apr 06 23:55:42 1992"
%d	Monatstag ("01" bis "31")
%H	Amerikanische Stundendarstellung ("01" bis "12")
%I	Stunde auf Europäisch ("00" bis "23")
%j	Tag im Jahr ("001" - "366")
%m	Monat ("01" - "12")
%M	Minute ("00" - „59")
%p	"AM" oder "PM"
%S	Sekunde ("00" - "61")
%U	Woche im Jahr ("00"-"53"), mit Sonntag als ersten Wochentag
%w	Wochentag (von "0" für Sonntag bis "6" für Samstag)
%W	Woche im Jahr ("00"-"53"), mit Montag als ersten Wochentag
%x	Datum, z. B. "Apr 06 1992"
%X	Uhrzeit, z. B. "23:55:42"

`%y` Jahr im Jahrhundert, z. B. `"92"`
`%Y` Vollständige Jahreszahl, z. B. `"1992"`
`%Z` Name der Zeitzone (hier nicht implementiert)
`%%` Prozentzeichen

Beispiel:

```
#include <stream.h>
#include <time.h>

void main()
{
  char s[80];
  time_t t = time(0);

  strftime(s, 80, "Es ist jetzt %X am %d. %m. %Y.\n",
  localtime(&t));
  cout « s;
}
```

gibt so etwas aus wie

Es ist jetzt 10:47:11 am 07. 04. 1995.

asctime

```
char *asctime (const struct tm *t)
```

ist eine Sparversion von `"strftime"` und wandelt die Zeit aus `"*t"` in eine Zeichenkette der Form `"Tue Apr 07 01:03:42 1992"` um. Diese Zeichenkette wird in einem internen Puffer abgelegt und ein Zeiger darauf zurückgegeben.

ctime

```
char *ctime(const time_t *t)
```

ist identisch mit `"asctime(localtime(t))"`, wandelt also einen `"time_t"`-Wert in eine Stringdarstellung um.

2. Bibliotheken für C++

2.1 Ein- und Ausgaben in C++: <stream.h> und <iostream.h>

2.1.1 Ausgabe

2.1.1.1 „ostream" und der Operator „«"

Bekanntlich ist in C++ alles viel schöner, besser und leichter als in ANSI C, und so ist es auch bei der Ein- und Ausgabe von Daten. Die dazu nötigen Definitionen stehen in der Includedatei „*<stream.h>*", während „*<iostream.h>*" nur so etwas wie ein Aliasname für erstere Datei ist und lediglich ein `"#include <stream.h>"` enthält. Genaugenommen ist „*<stream.h>*" die inzwischen veraltete Bezeichnung (aus dem 1.0-Standard) und „*<iostream.h>*" laut 2.0-Standard korrekter, aber letzten Endes ist das ja egal, denn die beiden Bezeichnungen sind absolut äquivalent. In diesem Handbuch wird konsequent „*<stream.h>*" benutzt.

In diesen Dateien wird unter anderem die Klasse `"ostream"` definiert, die für Ausgaben aller Art zuständig ist. Es gibt drei Standard-Objekte dieses Typs:

`cout` für gewöhnliche Ausgaben
`cerr` für Fehlermeldungen, die sofort ausgegeben werden
`clog` für gepufferte Fehlermeldungen

MaxonC++ nimmt sich die Freiheit, daß `"cerr"` und `"clog"` unterschiedliche Namen für ein und dasselbe Objekt sind.

Auf dieser Klasse `"ostream"` ist vor allem der Operator `"«"` mehrfach überladen. Der linke Operand ist dabei jeweils ein `"ostream"`-Objekt, über das die Ausgabe erfolgen soll, z. B. `"cout"`, und der rechte Operand ist ein Ausdruck eines Standard-Typs. Im einzelnen sind folgende Operatorfunktionen als Member definiert:

ostream &operator « (int)
ostream &operator « (unsigned)
ostream &operator « (long)
ostream &operator « (unsigned long)
ostream &operator « (long long)
ostream &operator « (unsigned long long)
ostream &operator « (unsigned char)
ostream &operator « (signed char)
ostream &operator « (char)
ostream &operator « (const char *)
ostream &operator « (float)

ostream &operator « (double)
ostream &operator « (long double)
ostream &operator « (const void*)
ostream &operator « (form &)
ostream &operator « (ios&(*f)(ios&))

Wie unschwer zu erkennen ist, gibt es für jeden numerischen Datentyp eine Ausgabefunktion (die Typen `"short"` und `"unsigned short"` werden gemäß den C++-Typregeln nach `"int"` umgewandelt und mit der entsprechenden Funktion ausgegeben). `"signed char"`- und `"unsigned char"`-Werte werden hier als Zahlen interpretiert, während bei `"char"` ein Zeichen ausgegeben wird. Bei einem Zeichenkettenargument (`"const char*"`) wird natürlich der entsprechende String ausgegeben, während bei einem beliebigen anderen Pointer („const void*") die Speicheradresse als hexadezimale Zahl mit vorangestelltem `"0x"` ausgegeben wird. Das kann natürlich zu Überraschungen führen, wenn man die Adresse einer `"char"`-Variablen ausgeben will:

```
#include <stream.h>

void main()
{
int i;
char c;

cout « &i;          // gibt Adresse von "i" hexadezimal aus
cout « &c;          // verunglückte String-Ausgabe
}
```

Hier versucht der Compiler, der nun einmal hinreichend dämlich ist, `"c"` als String zu interpretieren, und das Programm gibt alle Zeichen von `"c"` bis zum ersten im Speicher folgenden Nullbyte aus.

Außerdem stoßen die beiden letzten Funktionsdeklarationen in der obigen Liste bei Ihnen vielleicht auf Unverständnis. Die Funktion

```
ostream &ostream::operator « (form &)
```

stellt über die Klasse `"form"` eine formatierte Ausgabe im C-Stil zur Verfügung (siehe auch 2.1.1.3), und

```
ostream &ostream::operator « (ios&(*f)(ios&))
```

wendet eine Funktion auf eine Basisklasse von `"ostream"` an (2.1.1.3).

Das Ergebnis all dieser Funktionen ist stets eine Referenz auf das `"ostream"`-Objekt, so daß man mehrere Ausgaben aneinander hängen kann, z. B.:

```
int i=0x2a;
cout « "Der Wert von i ist: " « i « '\n';
```

Auch wenn die Werte natürlich in der Reihenfolge ausgegeben werden, in der man sie hinschreibt, kann ihre Auswertungsreihenfolge durchaus eine andere sein:

```
int i = 40;

cout « ++i « ++i;
```

Die Ausgabe kann hier entweder `"4142"` oder `"4241"` sein (die Ausgabe von Zahlen erfolgt ohne trennende oder führende Leerzeichen), denn voll ausgeschrieben entspricht dieser Ausdruck

```
cout.operator«(++i).operator«(++i)
```

und die Auswertungsreihenfolge in Ausdrücken ist in C++ nur in bestimmten Ausnahmefällen vorgeschrieben.

2.1.1.2 Basen und Manipulatoren

Die Klasse `"ostream"` hat noch eine Basisklasse namens `"ios"`. Aus bestimmten Gründen wird `"ostream"` sogar virtuell davon abgeleitet, aber das interessiert momentan nicht so unbedingt. Jedenfalls existieren ein paar Funktionen, nämlich `"dec"`, `"hex"`, `"oct"`, `"endl"` und `"flush"`, die als Parameter eine Referenz auf ein solches `"ios"`-Objekt erwarten und dieselbe Referenz als Ergebnis liefern. Da eine abgeleitete Klasse alle Eigenschaften ihrer Basisklassen erbt, kann man diese Funktionen natürlich auch auf die Klasse `"ostream"` anwenden.

Die drei Funktionen

```
ios &dec (ios &s)
ios &hex (ios &s)
ios &oct (ios &s)
```

setzen die Zahlbasis in einem Strom-Objekt. Nach einem Funktionsaufruf wie

```
hex(cout)
```

werden in Zukunft über `"cout"` alle ganzzahligen Werte hexadezimal, aber ohne vorangestelltes `"0x"`, ausgegeben. Die Ausgabe von Fließkomma- oder Stringwerten wird dadurch nicht beeinflußt.

Analog schaltet die Funktion `"oct"` auf oktale Zahldarstellung um, und `"dec"` stellt wieder den dezimalen Modus her.

Ein Beispiel:

```
#include <stream.h>

void main()
{
```

```
    int i = 42;
    hex(cout);
    cout « "hexadezimal: " « i « "\n";
    dec(cout);
    cout « "dezimal:     " « i « "\n";
    oct(cout);
    cout « "oktal:       " « i « "\n";
    cout « "Es gilt die jeweils zuletzt gewählte Einstellung: "
    « i « "\n";
}
```

Das Programm gibt aus:

hexadezimal: 2a
dezimal: 42
oktal: 52

Es gilt die jeweils zuletzt gewählte Einstellung: 52

Wie Sie an der letzten Ausgabe erkennen können, gelten die so vorgenommenen Einstellungen auch über eine Ausgabezeile hinaus, nämlich so lange, bis wieder ein anderer Modus gewählt wird.

Ein- und Ausgabedateien, und ein C++-Strom repräsentiert im Grunde genommen nichts anderes, werden oft mit einem Puffer beschleunigt (siehe auch 1.2.6.2). Die Funktion

```
ios &flush (ios &s)
```

löscht den Schreibpuffer eines Ausgabestroms, indem sie seinen Inhalt ausgibt. Wenn der Strom keinen Puffer hat, hat die Funktion keinerlei Wirkung. Maxon C++ weist den Standard-Ausgabeströmen als Default zwar keinen Puffer zu, aber auf einigen Systemen ist nach einer Ausgabe ein `"flush(cout)"` bzw. `"flush(clog)"` nötig, um die Ausgabe wirklich zu schreiben.

Natürlich kann man diese Funktion auch so benutzen:

```
flush(cout « "Ausgabe");
```

...oder so:

```
flush(cout) « "Ausgabe";
```

Der erste Ausdruck gibt erst etwas aus und wendet dann `"flush"` an, die zweite Anweisung schreibt erst einen etwaigen Pufferinhalt und macht dann eine weitere Ausgabe, die aber gegebenenfalls natürlich zunächst in einem Puffer aufbewahrt wird.

Last not least gibt es noch die Funktion

```
ios &endl (ios &s)
```

Sie gibt einen Zeilentrenner in einen Strom aus und ruft anschließend `"flush"` auf:

```
cout « "Das Ergebnis ist: " « Resultat;
```

```
endl(cout);
```

oder

```
endl(cout « "Das Ergebnis ist: " « Resultat);
```

entspricht

```
cout « "Das Ergebnis ist: " « Resultat;
flush(cout « "\n");
```

Diese kleinen Funktionen sind zwar ganz praktisch, aber ihre Handhabung und der damit verbundene Schreibaufwand entsprechen, wie auch die Übersichtlichkeit, doch eher dem C-Niveau als einem angenehmen C++-Stil. Deshalb gibt es die Funktion

```
ostream &ostream::operator « (ios&(*f)(ios&))
```

Wenn man die Adresse einer Funktion mit der passenden Signatur in ein `"ostream"`-Objekt „schreibt", wird sie darauf ausgeführt:

```
cerr « hex
```

ist damit äquivalent zu

```
hex(cerr)
```

Das eröffnet dem Programmierer reizvolle Möglichkeiten:

```
cout « hex « 26731 « endl;
```

schaltet die Standard-Ausgabe auf hexadezimale Ausgabe um, schreibt dann in eben diesem Modus eine Zahl und führt `"endl"` aus. Auf diese Weise wird die Handhabung solcher Funktionen doch erheblich angenehmer und intuitiver.

2.1.1.3 Formatierte Ausgabe

Die Klasse `"form"` stellt eine formatierte Ausgabe im Stil von `"printf"` zur Verfügung. Sie hat einen Konstruktor, dessen Parameter demgemäß eine nicht ganz unzufällige Ähnlichkeit mit denen von `"printf"` haben:

```
form(const char *fmt, ...)
```

In der Formatzeichenkette können sämtliche Umwandlungsoperationen und Optionen von `"printf"` (siehe 1.2.4) benutzt werden, und an Stelle der Ellipse `"..."` sind die entsprechenden zusätzlichen Argumente zu setzen. Man kann sich ein solches `"form"`-Objekt erzeugen und mit dem gewöhnlichen Operator `"«"` in einen `"ostream"` ausgeben:

```
#include <stream.h>

void main()
{
int i=42;
char s[ ] = "die große Frage";

form f("Die Antwort auf %s ist %d.\n", s, i);

cout « f;
}
```

In dem Objekt `"f"` wird hier der String

```
"Die Antwort auf die große Frage ist 42.\n"
```

abgespeichert und kann beliebig oft ausgegeben werden. Gebräuchlicher ist aber wohl die Möglichkeit, ein namenloses, temporäres Objekt auszugeben:

```
cout « form("Hex: %x", 42) « endl;
```

Auf diese Weise stehen auch bei den C++-Streams sämtliche Formatierungsmöglichkeiten von `"printf"` zur Verfügung. Der Nachteil ist natürlich, daß auch `"form"` eine Funktion ohne Netz und doppelten Boden ist und der Programmierer deshalb selbst darauf zu achten hat, daß die Argumente stimmen.

2.1.1.4 Ausgabe binärer Daten

Die Standardausgabe ist in der Regel ein Fenster, oder sie wird auch manchmal in eine (Text-)Datei umgelenkt. Deshalb wird man normalerweise über `"cout"`, `"cerr"` und `"clog"` nur formatierte Textdaten ausgeben. Man kann einen `"ostream"` aber auch an eine Datei anbinden, und dann wird man durchaus manchmal auch binäre Daten schreiben wollen. Deshalb gibt es die beiden Memberfunktionen

```
ostream &ostream::put (char c)
ostream &ostream::write (const char *p, int n)
```

Erstere Funktion schreibt ein Zeichen in einen Ausgabestrom, letztere die ersten `"n"` Bytes ab Adresse `"p"`. Daß `"p"` hier ein Zeiger auf `"char"` ist, sollte Sie nicht weiter beunruhigen, denn natürlich können Sie beliebige andere Zeiger nach `"char*"` casten.

Einige Beispiele:

```
cout.write("Hello, World!",5)
```

schreibt die Zeichenfolge `"Hello"` nach `"cout"`, und

```
double d = 3.14159;
cout.write((char*) &d, sizeof(double));
```

schreibt die `"double"`-Zahl `"d"` in denselben Strom, aber nicht als lesbare, formatierte Zahldarstellung, sondern in Form von (hier)acht Bytes.

Das erste Beispiel ist durchaus sinnvoll (manchmal möchte man wirklich nur einen Teil eines Strings ausgeben, oder man hat eine Zeichenkette vorliegen, bei der das Nullbyte am Ende fehlt), während das letztere wirklich nur bei Dateioperationen einen sittlichen Nährwert hat.

2.1.2 Eingabe

2.1.2.1 Texteingaben mit „»“

Was Krupp in Essen, wir im Trinken und `"ostream"` bei der Ausgabe, ist die Klasse `"istream"` für die Eingabe von Daten. Es gibt ein Standard-Objekt dieser Klasse namens `"cin"`, das in „*<stream.h>*“ deklariert wird und die Standard-Eingabe eines Programms darstellt, also normalerweise die Eingaben des Benutzers.

Die folgenden Operatoren sind als Member von `"istream"` deklariert:

istream & operator » (char s[ ])
istream & operator » (char &c)
istream & operator » (signed char &sc)
istream & operator » (unsigned char &uc)
istream & operator » (short &s)
istream & operator » (unsigned short &us)
istream & operator » (int &i)
istream & operator » (unsigned &u)
istream & operator » (long &l)
istream & operator » (unsigned long &ul)
istream & operator » (long long &vl)
istream & operator » (unsigned long long &uvl)
istream & operator » (float &f)
istream & operator » (double &d)
istream & operator » (long double &ld)

Linker Operand ist jeweils ein Objekt der Klasse `"istream"`, also insbesondere die Variable `"cin"`, und eine Referenz auf genau dieses Objekt ist auch der Rückgabewert dieser Operatorfunktionen. Als rechter Operand kann, wie Sie der imposanten Auflistung von Funktionen unschwer entnehmen können, ein L-Wert eines beliebigen numerischen Standardtyps (einschließlich `"char"`) oder eine Stringvariable dienen. Die Operatorfunktion liest dann die benötigte Anzahl von Zeichen aus dem Eingabestrom, wandelt sie in den gewünschten Datentypen um (z. B. eine Ziffernfolge in eine ganze Zahl) und weist das Resultat dem rechten Operanden zu.

Ein Beispiel:

```
int i;
char c;
char s[50];

cin » c » i » s;
```

Bei der Eingabe

```
0815-4711 42
```

wird "c" das Zeichen "0" zugewiesen, "i" bekommt den Wert "815" und in "s" wird die Zeichenkette "-4711" abgelegt, während die Zeichen "42" vorerst unbeachtet bleiben und vielleicht bei der nächsten Eingabe Beachtung finden. Liest man auf diese Weise ein "char", werden zuerst Leerzeichen, Zeilentrenner und sonstige Leerraum-Zeichen überlesen. Beim Lesen von Strings werden ebenfalls Trennzeichen überlesen, und es wird bis ausschließlich zum nächsten nachfolgenden Trennzeichen gelesen. Somit wirkt also - zumindest beim Operator "»" - auch ein Leerzeichen als Trennung zwischen zwei Strings.

Auch die Klasse "istream" kann nicht hellsehen, und deshalb weiß die Eingabe-Operatorfunktion "»", wenn sie auf eine Stringvariable angewandt wird, nicht, wie lang dieser String ist. Das kann zu üblen Abstürzen führen, wenn der Anwender bei

```
char input[20];

cin » input;
```

mehr als 20 Zeichen eingibt. Um dies halbwegs sicher zu gestalten, liest die Funktion in Maxon C++ maximal "STREAM_MAXSTRING" Zeichen in Stringvariablen, und zwar einschließlich des Nullbytes am Ende. Diese Konstante wird in „*<stream.h>*" als 80 definiert - ein kanonischer Wert für eine vernünftige Zeilenlänge. Leider handelt es sich dabei um eine Spezialität von MaxonC++ und nicht um einen wirklichen Standard.

Ganzzahlige Werte können wahlweise dezimal, oktal (mit führender Null) oder hexadezimal (mit "0x" oder "0X" am Anfang) eingegeben werden. Fließkommazahlen werden natürlich nur dezimal akzeptiert und können einen Exponententeil (mit "e" oder "E" eingeleitet) haben.

2.1.2.2 Eingabefunktionen auf „istream"

Die Memberfunktion "read" liest aus einem "istream" eine beliebige Anzahl von Bytes und legt sie in einem Zielvektor ab. Damit ist sie das Gegenstück zur ANSI-Funktion "fread". Sie ist definiert als

```
istream & istream::read(char buf, int n)
```

Auch hier sollte es Sie nicht weiter beunruhigen, daß der Zielvektor "buf" formal ein "char"-Array ist, denn man kann natürlich jeden Pointer in ein "char*" umwandeln. "n" ist die Anzahl

der Bytes, die gelesen werden sollen, und muß natürlich positiv sein. Das Ergebnis der Funktion ist wie immer eine Referenz auf das `"istream"`-Objekt.

Drei weitere Funktionen, die allesamt `"get"` heißen, lesen jeweils ein Zeichen aus dem Eingabestrom und sind folgendermaßen definiert:

```
istream & istream::get (char &c)
istream & istream::get (unsigned char &c)
int  istream::get ()
```

Die beiden ersten Funktionen lesen das Zeichen in eine `"char"`-Variable und liefern die übliche Referenz als Ergebnis. Sie sind gleichermaßen auf `"char"` und `"unsigned char"` definiert. Die dritte Funktion expandiert das Zeichen nach `"int"` und liefert es als Ergebnis.

Der Unterschied zwischen

```
cin » c
```

und

```
cin.get(c)
```

besteht darin, daß `"get"` keinen Leerraum überspringt, also zeichenweise die vollständige Eingabe einschließlich aller Leerzeichen und Zeilentrenner liest.

Oft merkt man, daß man genug gelesen hat, erst, nachdem man schon zu viel gelesen hat. In diesem Fall hilft die Funktion `"putback"`, die ein bereits gelesenes Zeichen in den Eingabestrom zurückstellt:

```
istream & istream::putback(char c)
```

Auf diese Weise kann aber immer höchstens ein Zeichen gepuffert werden.

Besonders beim Lesen aus Dateien muß man irgendwie wissen, wann man am Ende der Datei angelangt ist. Deshalb bietet `"istream"` auch eine Memberfunktion, die eben dieses testet:

```
int istream::eof()
```

Das Ergebnis dieser Funktion ist von Null verschieden, wenn das Dateiende überschritten wurde. Damit entspricht die Funktion `"feof"` aus „*<stdio.h>*“.

Nun fehlt noch eine Funktion, die eine Textzeile liest. Hier ist sie:

```
istream &getline(char *buf, int len, char Delim = '\n')
```

Es werden maximal `"len"`-1 Zeichen in den Vektor `"buf"` gelesen. Die Funktion bricht ab, sobald ein Trennzeichen `"Delim"` gelesen wird - als Default ist dies der Zeilentrenner. Dieses Trennzeichen wird zwar aus dem Eingabestrom gelesen und nicht wieder zurückgestellt, aber nicht in `"buf"` abgelegt.

2.1.3 Weitere Funktionen

2.1.3.1 Die Schnittstelle zum ANSI C-Dateisystem

Oben wurde bereits angedeutet, daß die `"stream"`-Funktionen von MaxonC++ im wesentlichen verkleidete ANSI-C-Funktionen sind. Diese Schnittstelle kann der Programmierer durch die Funktion `"getstream"` der Klasse `"ios"` nutzen:

```
FILE * ios::getstream()
```

Diese Funktion liefert als Ergebnis einen Zeiger auf den ANSI C- Filedeskriptor. Damit lassen sich sämtliche Funktionen aus „*<stdio.h>*" auch auf C++-Streams benutzen.

Außer den hinlänglich bekannten Objekten `"cout"`, `"cin"` und so weiter kann man auch selbst Objekte der Klassen `"istream"` und `"ostream"` erzeugen. Zu diesem Zweck haben die beiden genannten Klassen jeweils einen Konstruktor, der als Argument einen Dateizeiger erwartet.

Ein Beispiel:

```
#include <stream.h>
#include <stdio.h>

void main()
{
   FILE *f = fopen("test", "r");                    // C-File öffnen
   if (f)
   {
      istream is(f);                                // eigenes "istream"-
Objekt
      char s[80];

      is » s;                                       // liest einen String

      fclose(f);                                    // Datei wieder schließen
   }
}
```

`"istream"` und `"ostream"` haben keine Destruktoren, die die Dateien nachher automatisch schließen. Schon allein deshalb ist es nicht unbedingt sinnvoll, auf diese Weise mit Dateien zu arbeiten - mit den Klassen und Funktionen aus „*<fstream.h>*" geht das viel einfacher.

2.1.3.2 Fehlererkennung

Mit einer etwas eigenartigen Funktion bieten die Strom-Klassen eine äußerst intuitive Möglichkeit, Fehler in Dateien zu erkennen: In der Basisklasse `"ios"` gibt es eine Konvertierungsfunktion

```
ios::operator void*()
```

Diese Funktion liefert `"0"`, wenn das Dateiende überschritten oder ein anderer Fehler bemerkt wurde, und andernfalls einen Zeiger auf das Objekt selbst. Das sieht zunächst seltsam aus, ist aber

um so einfacher anzuwenden: Damit kann man Objekte und Ausdrücke des Typs `"ios"` oder davon abgeleiteter Klassen direkt in Abfragen einsetzen, z. B. so:

```
cin » s;
if (cin)
{ usw.
```

...oder auch so:

```
 char s[80];

while (cin » s)
cout « s;
```

Solche Ausdrücke sind logisch wahr, wenn kein Fehler aufgetreten und das Dateiende nicht erreicht ist. Natürlich kann man auf solche Abfragen auch die logischen Operatoren `"!"`, `"&&"` und `"||"` anwenden.

Kanonisch wäre natürlich gewesen, statt dieses komischen `"void"`-Zeigers eine Konvertierungsfunktion nach `"int"` oder einem anderen numerischen Typ zu implementieren, denn ein Zeiger als Bedingung ist doch ein wenig gewöhnungsbedürftig. Das ist aber nicht möglich, weil die Standard-Operatoren `"»"` und `"«"` auf ganzzahligen Werten definiert sind und Ausdrücke wie `"cout « n"` damit mehrdeutig würden.

2.2 Ströme und Dateien: <fstream.h>

2.2.1 Klassen für Dateien

Die Standard-Ein- und Ausgabeströme sind zwar ganz nett, aber natürlich braucht man daneben auch „richtige" Dateien, und es wäre angenehm, wenn man alle Features aus „*<stream.h>*" auch mit beliebigen Dateien benutzen könnte. Deshalb gibt es die Includedatei „*<fstream.h>*", in der Klassen und Funktionen für Dateien definiert werden.

In *<fstream.h>* werden drei für Sie interessante Klassen definiert:

- `"ifstream"` für Eingaben aus Dateien,
- `"ofstream"` für Ausgaben in Dateien,
- `"fstream"` für Ein- und Ausgaben.

`"ifstream"` ist von `"istream"` (und damit indirekt von `"ios"`) abgeleitet und erbt damit alle in 2.1.2 vorgestellten Eigenschaften. Entsprechend wurde `"ofstream"` von `"ostream"` abgeleitet, während `"fstream"` - ein interessantes Beispiel für Mehrfachvererbung - gleichzeitig `"istream"` und `"ostream"` als Basisklassen hat.

Die drei neuen Klassen haben außer diesen ererbten Funktionen auch noch einen Satz von elementaren Dateifunktionen, etwa Öffnen und Schließen von Dateien. Diese gemeinsamen Eigenschaften werden in der Klasse `"fstreambase"` zusammengefaßt, die ebenfalls Basisklasse von `"ifstream"`, `"ofstream"` und `"fstream"` ist.

2.2.2 Dateien öffnen und schließen

Die drei oben vorgestellten Datei-Stromklassen besitzen jeweils einen Default-Konstruktor. Wird aber ein Klassenobjekt damit erzeugt, ist es zunächst an keine physikalische Datei „angebunden", und alle Ein- und Ausgabeoperationen führen unabwendbar zu einem Absturz. Deshalb muß zunächst die Member-Funktion `"open"` der gemeinsamen Basisklasse `"fstreambase"` aufgerufen werden:

```
FILE * fstreambase::open (const char *name, int mode)
```

Der erste Parameter steht, wie man sich denken kann, für den Dateinamen, während der zweite den Zugriffsmodus festlegt. Mögliche Werte dafür werden von einem Aufzählungstyp, der in der Klasse `"ios"` verborgen ist, definiert:

```
enum open_mode
{
in  = 1,          // Eingaben
out = 2,          // Ausgaben
app = 4           // Anhängen
}
```

Beim Modus `"out"` wird stets eine neue Datei erzeugt. Existiert bereits eine Datei gleichen Namens, so wird diese gelöscht.

Das folgende Programm öffnet eine Datei und schreibt eine Zeile hinein:

```
#include <fstream.h>

const char dateiname[ ] = "ram:Test";

void main()
{
ofstream os;
os.open(dateiname, ios::out);
os « "Dies ist die Datei " « dateiname « ".\n";
os.close();
}
```

Wie Sie sehen, haben wir oben bereits eine weitere Funktion aus der Klasse `"fstreambase"` benutzt, nämlich `"close"`. Diese Funktion schließt die Datei, an die der Strom gebunden ist. Ein überflüssiges `"close"` schadet nicht. Nach dem Schließen ist das Stream-Objekt wieder an keinerlei Datei angekoppelt, und man kann, wenn man will, erneut eine Datei öffnen.

Kommen wir noch einmal auf die Zugriffsmodi von `"open"` zurück: Die drei vorgegebenen Werte können durch bitweises „Oder" miteinander kombiniert werden. Beispielsweise steht

```
ios::in|ios::out
```

für Lesen und Schreiben zugleich, oder beim Modus

```
ios::out|ios::app
```

wird zwar aus der Datei gelesen, aber der Lesezeiger steht zunächst am Ende der Datei und muß dann z. B. mit `"seekg"` (siehe unten) zurückgesetzt werden.

2.2.3 Fehlererkennung

Selbstverständlich sollte man immer, wenn man eine Datei öffnen läßt, prüfen, ob das auch wirklich geklappt hat, bevor man etwas mit der Datei anstellt. Dazu dient wieder die normale Fehlerkontrolle mit der eigenartigen Konvertierungsfunktion aus 2.1.3.2. Diese ist nämlich immer logisch „falsch", wenn ein Strom an keine Datei angekoppelt ist.

Das folgende Programm demonstriert dies, indem es eine Datei zum Lesen öffnet und sie dann in eine andere Datei kopiert:

```
#include <fstream.h>

void main()
{
   ifstream is;
   is.open("s:startup-sequence", ios::in);

   if(is)
   {
      ofstream os;
      os.open("ram:copy_of_s-seq", ios::out);

      if (os)
      {
         char s[80];
         while (is.getline(s,80))
            os « s « "\n";
         os.close();
      }
      else
         cout « "Kann Ausgabedatei nicht öffnen";

      is.close();
   }
   else
      cout « "Kann Eingabedatei nicht öffnen";
}
```

2.2.4 Konstruktoren und Destruktoren

Natürlich geht in C++ alles viel einfacher als in C, und so ist es auch bei diesen Dateioperationen. Ich verschwieg Ihnen bisher, daß `"ifstream"`, `"ofstream"` und `"fstream"` jeweils nicht nur einen Default-Konstruktor haben, sondern auch einen solchen, dessen Parameter mit denen der `"open"`-Funktion identisch sind. Dadurch kann man die Definition eines Dateiobjekts mit dem Öffnen der Datei aufs einfachste kombinieren, z. B. so:

```
ofstream outf("C++:Ausgabe", ios::out);
```

Außerdem ist für diesen Konstruktor bei `"ofstream"` als Default-Argument für den zweiten Parameter `"ios::out"` definiert, so daß die Deklaration sich verkürzt zu:

```
ofstream outf("C++:Ausgabe");
```

Einen entsprechenden Konstruktor gibt es auch für die Klasse `"ifstream"`, nur natürlich mit `"ios::in"` als Default-Argument. Auch für `"fstream"` existiert ein solcher Konstruktor, hier aber ohne einen abkürzenden Default-Modus.

Auch nachdem man ein Objekt auf diese erzeugt hat, muß man natürlich zuerst prüfen, ob die Datei auch wirklich geöffnet werden konnte, bevor man irgendwelche Ein- oder Ausgaben tätigt.

Außerdem haben die drei File-Stream-Klassen jeweils einen Destruktor, der die an das Objekt angebundene Datei schließt, sofern dies nicht schon geschehen ist. Er macht das natürlich unabhängig davon, mit welchem Konstruktor das Objekt erzeugt wurde.

Als Beispiel folgt noch einmal das in 2.2.3 vorgestellte Programm, diesmal aber unter Ausnutzung der Kon- und Destruktoren und mit vereinfachter Fehlerbehandlung:

```
#include <fstream.h>

void main()
{ ifstream is("s:startup-sequence", ios::in);
  ofstream os("ram:another_copy_of_s-seq", ios::out);

  if(is && os)
  { char s[80];
    while (is.getline(s,80))
      os « s « "\n";
    os.close();
  }
  else
    cout « "Kann nicht beide Dateien öffnen.";
}
```

2.2.5 Puffer

Auch bei den Strömen gibt es wieder das alte Elend, daß Dateioperationen nervtötend langsam sind, wenn byteweise oder in ähnlich kleinen Happen gelesen oder geschrieben werden muß. Deshalb gibt es in der Klasse `"fstreambase"` (also der gemeinsamen Basisklasse der drei Stromklassen) zwei Funktionen, deren Funktion sich dem geneigten Leser möglicherweise intuitiv erschließt:

```
int fstreambase::buffer (unsigned s)
void fstreambase::flush ()
```

`"buffer"` verleiht einer Datei einen Puffer der Größe `"s"` und entspricht damit einem „*<stdio.h>*"-Aufruf wie

```
setvbuf(getstream(), 0, _IOFBF, s)
```

Man kann einen Puffer für eine Datei immer nur einmal setzen und danach seine Größe nicht mehr verändern. Der benötigte Speicherplatz wird automatisch angefordert und beim Schließen der Datei ebenso automatisch wieder freigegeben. Das Ergebnis von `"buffer"` ist übrigens Null, wenn dabei kein Fehler auftrat.

`"flush"` schreibt den Inhalt eines Schreibpuffers in die Datei.

2.2.6 Positionieren in Strömen

Von den grundlegenden Dateioperationen fehlen jetzt nur noch Operationen zum Ermitteln und Setzen der Dateiposition. Zunächst wäre da die Funktion

```
int fstreambase::seekg (long offset, int mode = ios::beg)
```

Hier gibt es drei verschiedene Modi, die in Form eines Aufzählungstyps versteckt in `"ios"` definiert werden:

```
enum seek_dir
{
  beg = -1, // absolute Position zum Dateianfang
  cur = 0,      // relative Verschiebung von der akt. Position
  end = 1   // (negativer) Offset zum Dateiende
}
```

Das folgende Programm macht dem Rechner ein `"x"` für ein `"u"` vor, indem es in einer Datei alle `"u"` und `"U"` durch `"x"` bzw. `"X"` ersetzt. Dazu braucht man natürlich ein `"fstream"`-Objekt, denn man muß ja gleichermaßen lesen (um die „U"s zu suchen) und schreiben (nämlich die `"X"`-e). Die Datei wird hier zeichenweise eingelesen, und immer, wenn ein `"U"` gefunden wurde, wird die Position mit `"seekg"` um ein Zeichen zurückgesetzt und dann eben dieses `"U"` überschrieben:

```
#include <fstream.h>

const char name[ ] = "ram:test";

void main()
{
  fstream f(name, ios::in|ios::out);

  while (f)
  {
    char c = f.get();                    // Zeichen lesen

    if (c == 'u' || c == 'U')            // "U" ?
    {
      f.seekg(-1, ios::cur);             // dann zurück
      f.put( c=='u' ? 'x' : 'X' );       // und "X" schreiben
    }
  }
}
```

Als letztes lernen Sie jetzt noch eine Funktion kennen, mit der Sie die aktuelle Dateiposition ermitteln können. Sie heißt `"tellg"`:

```
long fstreambase::tellg ()
```

und liefert, wenn sie auf einem `"ifstream"`-, `"ofstream"`- oder `"fstream"`-Objekt aufgerufen wird, die Position bezüglich des Dateianfangs in Bytes. Die erste Position hat wie immer die Nummer Null.

2.3 Ein paar wichtige Definitionen in <new.h>

In der Datei „*<new.h>*" steht eigentlich fast nichts: Zum einen wäre da der Datentyp `"size_t"`, der mit `"unsigned"` identisch ist und auch in diversen anderen Files definiert wird, außerdem die hinlänglich bekannte Konstante `"NULL"` und die folgende Funktion:

```
void (*set_new_handler(void(*)(void))) (void)
```

Man muß natürlich dreimal hinsehen, um diese Deklaration als Funktion zu erkennen. Einfacher wird's, wenn man die Deklaration mit einem `"typedef"` aufteilt:

```
typedef void (*ftype) (void);
ftype set_new_handler (ftype);
```

Also erwartet die Funktion `"set_new_handler"` als Argument einen Zeiger auf eine Funktion, die weder Parameter noch Ergebnistyp hat, und gibt auch einen solchen zurück. Sinn und Zweck dieser Funktion ist es, eine Handler-Funktion zu setzen, die Speichermangel bei `"new"` oder `"malloc"` handhabt.

Normalerweise ignoriert es die Laufzeitbibliothek, wenn bei `"new"` oder `"malloc"` kein Speicher angefordert werden konnte, und liefert dann lediglich einen Nullzeiger, mit dem das Programm dann gefälligst selbst klarkommen muß. Rein praktisch bedeutet das, daß hinter jedes `"new"`, `"malloc"`, `"calloc"` usw. so etwas wie eine `"if"`-Abfrage gehört, die eine solche fehlgeschlagene Speicheranforderung abfängt und das Programm dann irgendwie sinnvoll weiterführt oder sauber beendet.

Bequemer kann man dieses lösen, indem man mit `"set_new_handler"` eine Funktion setzt, die in Zukunft immer dann aufgerufen wird, wenn der Speicher nicht ausreicht. Nach Ausführung dieser Funktion (sofern dort nicht mit `"exit"` o. ä. ausgestiegen wird) versucht die Bibliothek noch einmal, den benötigten Speicher anzufordern, und ruft im Mißerfolgsfall erneut den Handler auf.

Wählt man einen Nullzeiger als Argument für `"set_new_handler"`, ist wieder die Standard-Einstellung aktiv, d. h. das System kümmert sich nicht weiter um den Speicherüberlauf. Das Ergebnis von `"set_new_handler"` ist in jedem Fall die bisher gültige Einstellung, also ein Zeiger auf eine zuvor gesetzte Handlerfunktion oder eben Null.

Die Funktion `"set_new_handler"` wird auch in Abschnitt 2.6.6 des C++-Tutorials erläutert. Das folgende Beispielprogramm fordert gnadenlos so lange Speicherblöcke an, bis der Handler `"coredumped"` aufgerufen wird.

```
#include <new.h>
#include <stream.h>

unsigned MemUsed = 0;

void coredumped()
{
   cout « "Speicherüberlauf nach " « MemUsed « " Bytes.\n";
   cout « "Programm steigt aus!!!\n";
   exit(42);
}

void main()
{
   const unsigned int size = 10000;

   set_new_handler(coredumped);

   for(;;)
   {
      char *cp = new char[size];
      MemUsed += size;
   }
}
```

__MEMFLAGS

```
extern unsigned int __MEMFLAGS;
```

Beim Operator **"new"** und in den C-Funktionen **"malloc"**, **"calloc"** usw. wird stets beliebiger Speicher alloziert und mit Nullbytes gefüllt. Dieses Verhalten kann man aber ändern: Die Variable **"__MEMFLAGS"** enthält die Flags, die von der Standardbibliothek an die Exec-Funktion **"AllocMem"** übergeben werden. Standardmäßig ist diese Variable mit dem Wert **0x10000**, also **"MEMF_CLEAR"**, initialisiert.

Wenn Sie mit MaxonC++ ein Programm entwickeln, daß später auch auf anderen Computern und Compilern laufen soll, sollten Sie unbedingt am Anfang ihres Programms **"MEMF_CLEAR"** auf Null setzen, um so die Initialisierung des reservierten Speichers mit Nullbytes zu unterdrücken, und vielleicht noch zusätzlich ein Tool wie *„MungWall"* verwenden. Dann können Sie sicher sein, daß diese kleine Inkompatiblität Ihnen beim Portieren keine Probleme bereiten wird.

Eine andere denkbare Anwendung von **"__MEMFLAGS"**: Wenn Sie hier eines der Flags wie **"MEMF_CHIP"** oder **"MEMF_PUBLIC"** eintragen, können Sie die Speicherklasse der nachfolgenden **"new"**- oder „**malloc**"-Aufrufe festlegen.

2.4 Ausnahmebehandlung: <exception.h>

class Exception

```
class Exception
 { public:
     virtual ~Exception() { }
 };
```

Wie im Tutorial in Kapitel 8.3 schon angedeutet, greift die Klasse `"Exception"` dem sehnlichst erwarteten ANSI C++-Standard vor und definiert eine Basisklasse für alle Ausnahmen. Wenn Sie Klassen für Ihre Ausnahmen definieren, können Sie die von dieser Klasse ableiten - müssen Sie aber nicht.

unexpected

```
void unexpected();
```

Diese Funktion wird aufgerufen, wenn in erier Funktion it Ausnahme-Deklaration eine unerwartete Exception geworfen wird (siehe auch Tutorial 8.6). Normalerweise wird man sie nicht „von Hand" aufrufen.

`"unexpected"` ruft die zuletzt mit `"set_unexpected"` gesetzte Funktion auf. Default ist `"terminate"`.

terminate

```
void terminate();
```

`"terminate"` wird normalerweise nur von `"unexpected"` aufgerufen und enthält selbst nur einen Sprung an die zuletzt mit `"set_terminate"` gesetzte Funktion, was defaultmäßig `"abort"` ist.

set_unexpected

```
Definiton: void (*set_unexpected(void(*)()))();
```

Wie oben bereits erwähnt, kann man mit `"set_unexpected"` festlegen, in welche Funktion `"unexpected"` einspringen soll. Als Ergebnis wird die bisher gültige Funktion zurückgegeben.

set_terminate

```
void (*set_terminate(void(*)()))();
```

Analog zu `"set_unexpected"` verändert diese Funktion das Verhalten von `"terminate"`. Lesen Sie dazu auch das Kapitel 8.6 im Tutorial.

2.5 Eine String-Library: <tools/str.h>

2.5.1 Die Klasse „String"

Im Unterverzeichnis *„tools"*, das für Hilfsbibliotheken aller Art vorgesehen ist, finden Sie eine Datei *„str.h"*, die Ihnen eine C++ angemessene Stringverwaltung beschert.

Im wesentlichen wird dort eine Klasse namens `"String"` deklariert, die dynamische Zeichenketten repräsentiert. Das bedeutet: Man deklariert lediglich eine Variable des Typs `"String"`, und die Bibliothek sorgt selbst dafür, daß stets genug - und, einmal abgsehen von einem gewissen Overhead, auch nie zuviel - Speicher dafür reserviert wird. Im Tutorial wird in Kapitel 4.2.3 eine ähnliche (aber weniger leistungsfähige) Library als Beispielprogramm vorgestellt und implementiert.

Die Klasse `"String"` besitzt vier verschiedene Konstruktoren:

```
String (const char *s = 0)
String (int l, char *s)
String (const String &S)
String (char c)
```

Der erste Konstruktor initialisiert einen String mit einer Zeichenkette (oder dem Leerstring als Default) und dürfte der bei Variablendeklarationen meistgebrauchte sein, z. B.

```
String s1, s2 = "Horrido!", s3 = 0, s4 = "";
```

Hier werden die Variablen `"s1"`, `"s3"` und `"s4"` jeweils als Leerstrings initialisiert, während in `"s2"` die Zeichenkette `"Horrido!"` abgespeichert wird. Dazu muß der Konstruktor natürlich Speicher anfordern und dies in `"s2"` vermerken. Um solchen Speicher wieder freizugeben, wenn die Stringvariable nicht mehr existiert, besitzt die Klasse `"String"` auch einen Destruktor.

Den zweite Konstruktor benötigt man hauptsächlich, wenn man der Bibliothek eigene Stringfunktionen hinzufügen will. Aus einem `"char[ ]"`-Objekt, das mit `"new"` erzeugt worden sein muß und mit einem String, der ausschließlich des abschließenden Nullbytes die Länge `"1"` hat, initialisiert sein muß, wird hier ein `"String"`-Objekt erzeugt. Dabei wird die Zeichenkette `"s"` nicht noch einmal kopiert, sondern der Zeiger darauf direkt in das Stringobjekt eingetragen.

Der dritte Konstruktor ist, wie man sieht, ein Copy-Konstruktor, und der letzte initialisiert das Objekt mit einem Zeichen, das als Zeichenkette der Länge `"1"` interpretiert wird. Damit sind

```
String s1 = "?"
```

und

```
String s1 = '?'
```

mehr oder weniger äquivalent.

2.5.2 Member-Funktionen

Mit den oben aufgeführten Konstruktoren gibt es eine Konvertierung von den ordinären Zeichenvektoren, mit denen sich der C-Programmierer herumschlagen muß, in die Stringklasse. Für Kompatibilität in die andere Richtung sorgt die Konvertierungsfunktion

```
String::operator char*() const
```

Sie wandelt ein `"String"`-Objekt in einen Zeiger auf eine Zeichekette um. Dabei sollte man sich aber hüten, diese Konvertierung auf temporäre Objekte anzuwenden, z. B. auf das Ergebnis der unten vorgestellten Operatorfunktion `"+"`, denn natürlich liefert sie lediglich einen Zeiger auf den ansonsten „geheimen" Speicherbereich, in dem der eigentliche Wert des Stringobjekts abgespeichert wird, und wenn die Stringvariable nicht mehr existiert oder einen neuen Wert zugewiesen bekommt, muß man stets damit rechnen, daß dieser Speicherbereich freigegeben und wieder überschrieben wird.

Es gibt zwei Zuweisungsoperatoren:

```
String & String::operator = (const String &S)
String & String::operator = (const char *s)
```

Die erste Funktion funktioniert, wie Sie sich denken können, analog zum Kopier-Konstruktor, gibt aber den Speicher, der für den bisherigen Variablenwert alloziert wurde, wieder frei. Die zweite ist eigentlich redundant, denn der Compiler ist natürlich durchaus clever genug, bei einer Wertzuweisung wie

```
String s1;
char c[ ] = "Frood";

s1 = c;
```

aus dem Zeichenvektor mit Hilfe des entsprechenden Konstruktors ein temporäres Objekt zu erzeugen und dieses dann mit der erstgenannten `"="`-Funktion in das Zielobjekt zu kopieren. Mit dieser zusätzlichen Funktion wird dies aber deutlich abgekürzt.

Ungefähr wie "strcat" in Standard-C, nur eben erheblich bequemer, funktioniert die Member-Funktion

```
String & String::operator += (const String &s),
```

die zwei Strings aneinander hängt: Nach

```
String s1 = "Moin ", s2 = "Moin!";

s1 += s2;
```

hat "s1" den Wert "Moin Moin!".

Sie wollen auf einen String zeichenweise zugreifen? Null Problemo, auch das geht, und zwar mit der folgenden Operator-Funktion:

```
char String::operator[ ] (int i) const
```

Sie liefert als Ergebnis das "i"-te Zeichen einer Stringvariablen, allerdings nicht als Referenz und damit auch nicht als L-Wert. Deshalb kann man mit Indizierung Zeichen aus einem String auslesen, aber keine einzelnen Zeichen verändern. Dies wurde aus Sicherheitsgründen so eingerichtet.

Um die Länge eines Strings festzustellen, können Sie die Standard-Funktion "strlen" benutzen, denn es existiert ja eine Konvertierung von "String" nach "char*". Angenehmer und schneller ist aber die folgende Member-Funktion:

```
int String::length() const
```

Sie liefert die Länge eines Strings, wobei das Nullbyte am Ende wie immer nicht mitgezählt wird. Aufgerufen wird sie natürlich in der Form

```
i = s1.length();
```

Zu guter Letzt wären da noch drei Funktionen, die irgendwie an alte BASIC-Tage erinnern:

```
String String::left (int n) const
String String::right (int n) const
String String::mid (int pos, int n) const
```

"left" liefert die ersten "n" Zeichen eines Strings, "right" die letzten "n". Falls der String kürzer als "n" ist, ist das Ergebnis mit dem Argument identisch. "mid" nimmt aus einem String "n" Zeichen ab der Position "pos", wobei letztere wie immer von Null an gezählt wird.

2.5.3 Der Operator „+" auf Strings

Das waren alle Funktionen, die als Member von `"String"` deklariert sind. Daneben gibt es noch einige global deklarierte Operatorfunktionen:

Die angenehmste ist sicherlich

```
String operator + (String s1, String s2)
```

Dieses `"+"` hängt, wie man es auch nicht anders erwartet, zwei Strings aneinander. Anders als bei `"strcat"` oder dem Operator `"+="` werden hier die Werte der Argumente keineswegs verändert, sondern es wird ein neues (temporäres) Objekt erzeugt, falls das Ergebnis nicht sogar direkt in das gewünschte Ziel geschrieben wird.

Da es sonst keine Addition auf Standard-Strings (d. h. char-Zeigern) gibt, kann dieser neue Plus-Operator auch dort angewendet werden:

```
#include <tools/str.h>
#include <stream.h>

void main()
{  char *s1 = "Grunz",
   *s2 = s1 + "wanzling";

   cout « s2;
}
```

Eigentlich ist dieses Beispiel korrekt, denn bei der Addition wird der `"+"`-Operator aus der String-Bibliothek benutzt und dessen Ergebnis, ein temporäres Objekt der Klasse `"String"`, mit der Konvertierungsfunktion nach `"char*"` gewandelt. Dabei wird aber der alte Fehler begangen, einen Zeiger auf ein temporäres Objekt zu verwenden. Wenn `"s2"` ausgegeben wird, existiert das temporäre Objekt (und mit ihm sein Zeichenvektor) schon lange nicht mehr, und die Ausgabe ist undefiniert.

So hätte es natürlich geklappt:

```
#include <tools/str.h>
#include <stream.h>

void main()
{  char *s1 = "Grunz";
   String s2 = s1 + "wanzling";

   cout « s2;      // Ausgabe für "String" existiert - siehe unten
}
```

Hier wird - je nach Compiler - die String-Summe entweder gleich im Zielobjekt `"s2"` konstruiert, oder das hier eingeführte temporäre Objekt wird mit dem Copy-Konstruktor in `"s2"` übertragen, bevor es destruiert wird. Also funktioniert `"+"` hier so, wie man es erwartet.

Deshalb soll hier noch einmal erwähnt werden, daß die Konvertierungsfunktion von `"String"` nach `"char*"` nur mit Vorsichtig und Bedacht zu benutzen ist, denn andernfalls erlebt man Überraschungen.

2.5.4 Vergleiche

Natürlich kann man String-Objekte mit der Funktion `"strcmp"` vergleichen. Mit den folgenden Operatoren geht das aber wesentlich bequemer:

```
int operator == (const String &s1, const String &s2)
int operator != (const String &s1, const String &s2)
int operator <  (const String &s1, const String &s2)
int operator >  (const String &s1, const String &s2)
int operator <= (const String &s1, const String &s2)
int operator >= (const String &s1, const String &s2)
```

Die beiden Operanden werden jeweils wie mit `"strcmp"` verglichen (ASCII-Reihenfolge), und das Ergebnis ist ein boolscher Wert.

Ein Mini-Beispiel:

```
#include <tools/str.h>
#include <stream.h>

void main()
{ String s1 = "Test", s2 = "Text";

  if (s1 < s2)
    cout « s1;
  else
    cout « s2;
}
```

Ein kleiner Wermutstropfen: Vergleiche wie

```
String s1;

if(s1 == "Nanu?") ...
```

sind nicht erlaubt, da mehrdeutig! Schließlich gibt es sämtliche Vergleichsoperationen auch auf Zeigern (und insbesondere Zeichenketten), und so weiß der Compiler oben nicht, ob er `"s1"` nach `"char*"` konvertieren und diese Adresse dann mit der des Strings „Nanu?" vergleichen, oder ob er aus der konstanten Zeichenkette mittels Konstruktoraufruf ein temporäres `"String"`-Objekt erzeugen und darauf den String-Operator `"=="` anwenden soll.

So geht's aber:

```
if (s1 == String("Nanu?"))
```

2.5.5 Ein- und Ausgabe

In den obigen Beispielen wurden mehrfach Objekte der Klasse `"String"` ausgegeben. Das ist ohne weiteres möglich, denn die Operatoren wurden - wie im Tutorial unter Abschnitt 4.2.2.7 beschrieben - entsprechend zusätzlich überladen:

```
class ostream& operator « (ostream&, const String&)

class istream& operator » (istream&, String&)
```

Die Ein- und Ausgabe von `"String"`-Objekten funktioniert damit genau wie die von gewöhnlichen `"char*"`-Zeichenvektoren.

2.6 Interna von MaxonC++: <streamdefs.h>

Jeder hat so seine kleinen Geheimnisse - aber MaxonC++ verrät sie hemmungslos. Na ja, zumindest gibt es in der Datei „*<streamdefs.h>*“ einige Definitionen, die über die interne Organisation der Dateiverwaltung von MaxonC++ Aufschluß geben.

In „*<stdio.h>*“ wird der enorm wichtige Datentyp **"FILE"** definiert, der nichts als ein Alias für **"struct stream"** darstellt. Dieser Strukturtyp wird in „*<streamdefs.h>*“ definiert, und zwar wie folgt:

```
struct stream
{ unsigned Filehandle;
  char UngetCh, UngetBuf;
  signed char Mode, Error;
  struct streambuffer *bufptr;
  struct {
    int f_freemem:1,
    f_closefile:1
    } flags;
};
```

Die Felder haben folgende Bedeutung:

Filehandle: Dies ist die AmigaDOS-Filehandle, die zur Datei gehört.

UngetCh: Ein Flag, das anzeigt, ob ein Zeichen mit **"ungetc"** zurückgestellt wurde.

UngetBuf: Der Puffer, in dem eben jenes Zeichen abgelegt wird.

Mode: In diesem Byte steht ist das Bit #0 gesetzt, wenn die Datei zum Lesen geöffnet wurde, und Bit #1 steht entsprechend für Schreibzugriffe.

bufptr: Wenn dieser Zeiger nicht Null ist, zeigt er auf eine **"streambuffer"**-Struktur, in der Informationen über das Puffern der Datei enthalten sind.

flags: Das Bit **"f_freemem"** (Nummer 7) dieses Byte-großen Bitvektors zeigt an, ob **"fclose"** die **"stream"**-Struktur aus einer gewissen internen Liste aushängen soll, und **"f_closefile"** wird gesetzt, wenn bei **"fclose"** die DOS-Datei „*Filehandle*“ geschlossen werden soll.

Die Struktur **"streambuffer"** wird normalerweise von den Funktionen **"setvbuf"** oder **"setbuf"** erzeugt und in die **"stream"**-Struktur eingehängt. Sie ist folgendermaßen aufgebaut:

```
struct streambuffer
{ stream *streamptr;
  short size, fill, pos;
  signed char mode, own;
  int (*read)(register streambuffer *a0, register void *d2,
    register unsigned d3);
  int (*write)(register streambuffer *a0,
    register void *d2, register unsigned d3);
  int (*flush)(register streambuffer *a0);
```

```
    int (*close)(register streambuffer *a0);
    void *buf;
};
```

Die Bedeutung der Member:

streamptr: Ein Rückverweis auf die "stream"-Struktur, deren Zeiger "bufptr" wiederum auf diese "streambuffer"-Strukt ur zeigt.

size: Puffergröße (in Bytes)

fill: Anzahl der momentan benutzten Puffer-Bytes

pos: Bei Lesepuffern die aktuelle Leseposition

mode: Ist positiv, wenn es sich um einen Schreibpuffer handelt, negativ, wenn der Puffer zum Lesen dient, und Null, wenn der Puffer aus irgendwelchen Gründen überhaupt nicht benutzt werden soll.

own: Ein Flag, das gesetzt wird, wenn "setvbuf" den für den Puffer "buf" benötigten Speicher selbst alloziert hat. Dadurch weiß die Laufzeitbibliothek beim Schließen der Datei, ob der Speicherbereich, auf den "buf" zeigt, freigegeben werden muß.

read: Ein Zeiger auf eine Funktion mit den angegebenen Register-Parametern. Sie wird bei Eingaben aus der gepufferten Datei aufgerufen und sollte sich wie die AmigaDOS-Funktion "Read" verhalten, einmal davon abgesehen, daß das erste Argument hier keine Filehandle, sondern ein Zeiger auf den "streambuf" ist und auch nicht in "d1", sondern in "a0" übergeben wird.

write: Die entsprechende Funktion für Ausgaben in Dateien.

flush: Ebenfalls ein Zeiger auf eine Funktion. Sie wird (wie man sich denken kann) aufgerufen, wenn auf der Datei "fflush" ausgeführt wird oder wenn sie geschlossen wird.

close: Diese Funktion wird beim Schließen der Datei aufgerufen und sollte insbesondere den vom Puffer belegten Speicher freigeben. Falls die "streambuffer"-Struktur dynamisch erzeugt wurde, sollte die "close"-Funktion auch diese löschen.

buf: Ein Zeiger auf den eigentlichen Puffer.

Die Sache mit diesen Zeigern auf Funktionen hätte man natürlich viel eleganter und einfacher mit Vererbung und virtuellen Memberfunktionen realisieren können, aber die Definitionen sollten auch im ANSI C-Modus benutzt werden können. Es steht dem Programmierer natürlich frei, diesen Defi-

nitionen einen objektorientierten Ansatz zu überlagern, so wie in *„stream.h"* das C++-typische Stromkonzept auf die ANSI-Dateiverwaltung aufgesetzt wird.

Was soll das ganze eingentlich? Nun, mit diesen Informationen (und vielleicht ein wenig Herumprobieren) ist es möglich, eigene Routine auf tiefem Niveau in die Ein-/Ausgaberoutinen der Maxon C++ Laufzeitbibliothek zu integrieren. Es ist ja nicht gesagt, daß eine Struktur **"streambuffer"** immer nur einen Puffer repräsentiert. Beispielsweise könnte man hier Routinen einhängen, die Ein- und Ausgaben in Intuition-Windows machen, und dann z. B. mit **"printf"** in einen Dialogfenster schreiben. Der Phantasie sind hier fast keine Grenzen gesetzt.

Übrigens: Wenn hinter einem **"stream"** gar keine wirkliche DOS-Datei, sondern ein **"streambuffer"** mit selbstgeschriebenen geheimnisvollen Funktionen steht, sollte man den Member **"Filehandle"** auf Null setzen. Dann wissen Funktionen wie **"fseek"** oder **"ftell"** Bescheid und versuchen erst gar nicht, auf dieser nicht existenten DOS-Datei zu operieren.

Um das Herumtrashen an der Dateiverwaltung zu erleichtern, gibt es die Funktion **"allocstream"**:

```
stream *allocstream(unsigned Handle)
```

Sie erzeugt ein neues Objekt des Typs **"stream"**, trägt **"Handle"** als Filehandle ein und vermerkt die Struktur in der internen Liste der Dateien. Dadurch wird der Strom z. B. am Dateiende automatisch geschlossen, und **"fflush(0)"** wirkt auch auf diesen Stream. Dementsprechend wird auch das Flagbit **"f_freemem"** gesetzt.

Falls nicht mehr genug Speicher für die Struktur frei ist, liefert **"allocstream"** wie üblich eine Null.

2.6 Funktionen für originelle Linker-Optionen: <linkerfunc.h>

In der Include-Datei „*<linkerfunc.h>*" finden Sie einige MaxonC++-typische Funktionen, die Sie vor allem dann brauchen, wenn Sie ohne Startup-Code linken (siehe auch Abschnitt 4.6 des Benutzerhandbuchs).

Jede Objektdatei kann in C++ dynamische Initialisierungen benötigen, also eine Funktion besitzen, die beim Programmstart unbedingt aufgerufen werden muß. Außerdem kann es natürlich statische Daten geben, für die am Programmende Destruktoren aufgerufen werden müssen. Der Linker generiert Funktionen, in denen alle diese Initialisierungs- bzw. Aufräumfunktionen augerufen werden:

```
void InitModules()
```

ruft alle Initialisierungen auf, und

```
void CleanupModules()
```

die Destruktoren. Wenn man sich einen Startup-Code selbst schreibt, sollte man am Anfang auf jeden Fall `"InitModules"` und am Programmende `"CleanupModules"` aufrufen.

Es kommt bisweilen vor, daß eine C++-Funktion nicht aus einem C++-Programm, sondern von irgendwoher aufgerufen wird, z. B. wenn man eine Shared Library in C++ geschrieben hat. Das gibt natürlich Probleme, wenn man in seiner Funktion „Small Data„ verwendet, denn dann vertraut die Funktion darauf, daß im Register A4 ein Zeiger auf den Datenhunk steht.

Was tun? Nun, man muß diesen Zeiger explizit laden, indem man ganz am Anfang der Funktion (also, jedenfalls vor jeglichem Zugriff auf statische Daten)

```
void GetBaseReg()
```

aufruft.

AMIGA

MaxonC++

Klassenbibliothek

EASY-OBJECTS

MAXON computer

Einführung

Wie arbeitet man mit der Klassenbibliothek? Der Grundaufbau und das Hauptprogramm jeder Applikation, die eine graphische Benutzeroberfläche besitzen soll, sieht immer ziemlich ähnlich aus.

Als Beispiel wird hier das unvermeidbare HelloWorld-Programm beschrieben, allerdings erfolgt die Ausgabe in einem größenveränderbaren Fenster.

Diese drei Include-Anweisungen importieren einen Großteil der Klassenbibliothek...

```
#include <classes/layouter/windows.h>
#include <classes/layouter/boopsigadgets.h>
#include <classes/exec/libraries.h>
```

Beginnen wir mit der Deklaration eines Fensters. Dieses wird von der Klasse `StandardWindowC` abgeleitet, eine Fensterklasse, deren Fenster größenveränderbar, verschiebbar und schließbar sind. Außerdem wird automatisch der Zeichensatz des Bildschirms für die Gadgets verwendet.

```
class HelloWorldWindowC : public StandardWindowC {
public:
   HelloWorldWindowC(GTIDCMPortC &, ScreenC &);
   ~HelloWorldWindowC();
private:
```

Wir verwenden ein Textausgabefeld, eine Geometrie dafür und einen Handler für das Schließsymbol des Fensters.

```
   LBTextC hello;
   GeometryC helloGeo;
   WindowCloseHandlerC wch;
};
```

Bei solchen neuen Fenstern ist der Konstruktor die wichtigste Methode. Darin wird das Layout aufgebaut und die Handler des Fensters aktiviert.

```
HelloWorldWindowC::HelloWorldWindowC(GTIDCMPortC &p, ScreenC &s)
```

Das Standardfenster ist schon so, wie wir es wollen. Hier könnte man aber z.B. noch eine Größe angeben, oder festlegen, in welchem Rand das Sizegadget sitzt oder einen Titel des Fensters oder ...

```
   : StandardWindowC(p,s,NULL,
     TAG_END),
```

Das Textausgabefeld wird ohne Ereignisbehandlung initialisiert (es kann auch in diesem Beispiel keine Ereignisse sinnvoll behandeln, denkbar wäre allerdings eine Helpfunktion o.ä).

```
     hello(NULL,*this,LAYOUT_AUTOSIZE,LAYOUT_AUTOSIZE,
     LBA_Text,"Hello World!",
     LBA_Adjust,LB_AdjustCenter,
     TAG_END),
```

Die Geometrie initialisieren wir derart, daß das Gadget oben, unten, rechts und links an den Rändern der Gruppe hängt, d.h. das Gadget wird den gesamten inneren Bereich des Fensters füllen.

```
helloGeo(hello,
 LAYOUT_GROUP,NULL,2,LAYOUT_GROUP,NULL,-2,
 LAYOUT_GROUP,NULL,2,LAYOUT_GROUP,NULL,-2),
```

Der Handler des Schließsymbols hängt sich durch die Initialisierung (wobei ja die Referenz auf das Fenster angegeben wird) selbst in die Handlerkette des `GTIDCM` Ports ein und wird dadurch aktiv. Dieser Schließhandler beendet einfach das Programm.

```
    wch( *this)
{
```

Das Gadget wird in die Liste der Gadgets eingehängt und die Geometrie in die Geometriegruppe, die den inneren Bereich des Fensters umfasst. Da wären theoretisch auch noch die 4 Ränder des Fensters möglich. Diese Bereiche sollten allerdings Spezialfällen vorbehalten sein (z.B. Scrollern im unteren und rechten Rand des Fensters).

```
  gadgets.add(hello);
  innerGeo.add(helloGeo);
}
```

Wichtig

Der Konstruktor schließt das Fenster, damit die Gadgets nicht vor dem Schließen des Fensters freigegeben werden. Dies könnte zu Problemen führen, wenn in dieser Zeit noch ein letztes Refresh-Event kommt.

Unbedingt merken!!

```
HelloWorldWindowC::~HelloWorldWindowC()
{
  close();
}
```

Sämtliche Libraries, die für den Betrieb der Klassenbibliothek mindestens benötigt werden, werden geöffnet.

```
LibraryBaseErrC GadToolsBase("gadtools.library",37);
LibraryBaseErrC UtilityBase("utility.library",37);
LibraryBaseErrC CxBase("commodities.library",37);
LibraryBaseErrC LayersBase("layers.library",37);
LibraryBaseErrC WorkbenchBase("workbench.library",37);
```

Das Hauptprogramm sieht eigentlich so sehr typisch aus:

```
int main()
{
```

Zuerst wird überprüft, ob alle Libraries geöffnet werden konnten. Eine Fehlermeldung ist durch die Verwendung der Klasse `LibraryBaseErrC` allerdings schon ausgegeben, das Programm braucht nur abgebrochen zu werden.

```
  if (!LibraryBaseC::areAllOpen())
    return 20;
```

Das folgende Objekt ist das Kernstück des Programms, es empfängt und verarbeitet alle Signale (und damit auch Nachrichten, Semaphoren etc).

```
  SignalsC sc;
```

Jedes Fenster will seinen Bildschirm genau kennen. Der Einfachheit halber wird hier der Standardbildschirm gesperrt, ein "echtes" Programm sollte natürlich etwas flexibler sein.

```
  PublicScreenC screen();
  if (!screen.lock(NULL))
    return 100;
```

Alle `IDCMP` Nachrichten aller Fenster können durch einen einzigen Nachrichtenport empfangen werden. Dieser wird in der Variante für Gadtools-Fenster verwendet da die Fensterklasse `StandardWindowC` davon indirekt abgeleitet ist.

```
  GTIDCMPortC port;
  sc.add(port);
```

Das Fenster wird initialisiert.

```
  HelloWorldWindowC window(port,screen);
```

Das Programm kann durch den folgenden Handler auch noch mit <Ctrl>+<C> abgebrochen werden.

```
  CtrlCHandlerC ctrlchandler;
  sc.add(ctrlchandler);
```

Nun wird das Fenster geöffnet. Wenn das schief geht, wird das Programm durch eine Ausnahmebehandlung einfach abgebrochen. In einem "echten" Programm sollte dieser Fall abgefangen sein.

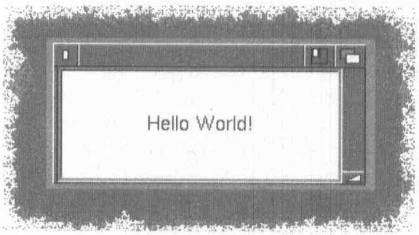
```
  window.open();
```

Nun wird die Signalbehandlung aktiviert. Das Programm befindet sich bis zum Ende in dieser Schleife und behandelt ankommende Signale.

```
  sc.loop();
```

Ende gut...

```
    return 0;
}
```



Wenn sie dieses Programm kompilieren und starten (es befindet sich in der Demo Schublade unter dem Namen HelloWorld.c), sehen sie ein kleines Fenster, in dem "Hello World!" ausgegeben wird. Sie können die Größe des Fensters verändern - der Text bleibt immer zentriert stehen. Beenden sie das Programm durch Drücken des Schließsymbols des Fensters.

Die Ausnahmebehandlung

Mit dem C++ Standard 3.0 wurden Ausnahmen (Exceptions) in die Sprache eingeführt. Diese Ausnahmen erleichtern die Fehlerbehandlung deutlich. EASY-OBJECTS verwendet die Ausnahmen allerdings nur für "echte" Fehler, d.h. bei Speichermangel oder Mangel anderer Ressourcen. Hingegen wird das Fehlschlagen des Öffnens einer Datei nicht als Ausnahme behandelt, schließlich ist es kein sehr seltener Fall, daß eine Datei geöffnet werden soll, die nicht existiert. In diesen und ähnlichen Fällen geben die entsprechenden Methoden Fehlercodes als Ergebnis zurück.

Die Ausnahmebehandlung der Bibliothek basiert auf einer Klassenhierarchie, die sämtliche möglichen Ausnahmen der Klassenbibliothek umfaßt. Einige Ausnahmen enthalten weitere Daten, um auf die Fehlerursache zurückzuschließen. Dabei werden grundsätzlich zwei Ausnahmearten unterschieden:

1. Ausnahmen, die durch eine falsche Verwendung der Klassen zurückzuführen sind. Dazu zählen z.B. fehlerhafte Layouts. Grundsätzlich sind diese Ausnahmen auf Programmfehler zurückzuführen und als Bug im Programm zu werten.

2. Ausnahmen, die durch fehlschlagende Ressourcenallozierung ausgelöst werden. Die meisten Versuche Speicher zu allozieren, z.B. für Systemstrukturen, führen zu Ausnahmen dieser Art, insbesondere werfen alle fehlschlagenden Konstruktoren Ausnahmen.

Die Klassenhierarchie enthält drei Schichten: alle Ausnahmen sind von der Klasse **`Exception`** abgeleitet. Diese abgeleiteten Klassen definieren funktionelle Teilbereiche, in die die konkrete Ausnahme fällt, z.B. finden sie alle Ausnahmen, die durch eine fehlschlagende Ressourceallozierung ausgeworfen werden, von der Teilbereichsklasse **`ResourceX`** abgeleitet. Von diesen Teilbereichsklassen sind nun direkt die konkreten Ausnahmen abgeleitet, diese enthalten zum Teil weitere Daten, um die konkrete Fehlersituation zu rekonstruieren.

Container und Cursor

Die Klassenbibliothek enthält verschiedene Klassen, deren Objekte andere Objekte sammeln können.

Die Exec Library stellt doppelt-verkettete Listen zur Verfügung, diese Listen stehen in zwei Varianten zur Verfügung: die Klasse **`ListC`** mit den Einträgen **`NodeC`** basiert auf der Struktur **`MinList`**, die Klassen **`EListC`** mit den Einträgen **`ENodeC`** basieren auf der Struktur **`List`**.

Weitere wichtige Container sind die dynamischen Arrays. Diese stehen sowohl in einer generischen Form, wie auch als Templates zur Verfügung. Von der Klasse **`gen_array`** bzw. **`array`** erben zwei Klassen, die auf diesen Arrays einige typische Listenoperationen ermöglichen, die Klassen **`gen_arraylist`** und **`arraylist`**.

Desweiteren gibt es einige Containerklassen für Spezialanwendungen, z.B. für Workbenchargumente und Tooltypes, die Liste öffentlicher virtueller Bildschirme und Taglisten.

EASY-OBJECTS verwendet ein einheitlichens Prinzip um die Objekte der Containerklassen zu verarbeiten. Zu jeder Containerklasse gehört eine passende Cursorklasse. Jeder Cursor stellt neben dem Konstruktor, dem sie mitteilen, welches Containerobjekt bearbeitet werden soll, zumindest 3 Methoden zur Verfügung: Zuerst einmal eine Methode `isDone()` zum Test, ob der Cursor schon am Ende des Containers steht, d.h. ob sämtliche Elemente bearbeitet wurden. Als nächstes eine Methode `next()`, durch die der Cursor auf das nächste Element des Containers gesetzt wird, und schließlich noch eine Methode `item()` mit dem auf das aktuelle Element, auf das der Cursor zeigt, zugegriffen wird.

Die meisten Cursor kennen noch eine Methode `first()` um den Cursor wieder auf den Anfang des Containers zurückzusetzen und die Listencursor (`ListCursorC` und `EListCursorC`) kennen noch die Methoden `last()` und `prev()`, um die Liste auch von hinten nach vorne zu durchsuchen.

Beispiel:

```
arraylist<int> zahlen;
arraycursor<int> zaehler;

zahlen.addTail() = 6;
zahlen.addTail() = 8;
zahlen.addTail() = 42;

while (!zaehler.isDone())
{
   cout << zaehler.item() << "\n";
   zaehler.next();
};
```

Dieses Beispiel legt die drei Zahlen 8, 6 und 42 auf eine Liste, die allerdings über ein dynamisches Array implementiert ist. Dann werden über die Verwendung des Cursors die drei Zahlen wieder ausgegeben.

Der Vorteil der Verwendung der Cursor liegt klar auf der Hand: Egal, ob Sie doppelt-verkettete Listen, dynamische Arrays, Workbenchargumente, Tooltypes oder Taglisten verarbeiten, der Zugriff erfolgt immer identisch und Sie können solche Schleifen sehr viel schneller und weniger fehleranfällig programmieren.

Das Handlerkonzept

Auftretende Ereignisse werden durch Signale angezeigt. Diese Ereignisse können sehr unterschiedlicher Natur sein, es kann z.B. das Signal <CTRL>+<C> direkt gesetzt werden, oder eine Nachricht an einem Nachrichtenport eintrudeln.

Um diese Ereignisse zu verarbeiten, werden Signalhandler eingesetzt. Die Klasse **SignalsC** verwaltet eine Liste von Signalhandlern (ein neuer Handler wird mit der Methode **SignalsC::add** angehängt), und kann diese Signale verarbeiten, indem bei einem ankommenden Signal jeder Handler gefragt wird, ob er das Signal verarbeiten kann. Ist dem so, wird eine Behandlungsmethode des Handlers aufgerufen, die das Signal verarbeitet – z.B. die nächste Nachricht vom Port holt, diese irgendwie bearbeitet und an den Sender zurückschickt.

Das folgende Beispiel benutzt einen Handler für <Ctrl>+<C>. Dieser Handler reagiert auf das Setzen des entsprechenden Signals.

```
int main()
{
   SignalsC sc;

   CtrlCHandlerC ctrlc();
   sc.add();

   sc.loop()

   return 0;
}
```

Bleibt die Frage, warum das Programm beendet wird, wenn <Ctrl>+<C> gedrückt wird. Nun, die Methode, die das Signal verarbeitet, kann als Ergebnis zurückgeben, daß die Methode **SignalsC::loop** verlassen werden soll. Genau dies und nichts mehr macht die Klasse **CtrlCHandlerC** und damit wird das Programm auch beendet.

Für Nachrichtenports funktioniert dieses Handlerkonzept auch. Die meisten Nachrichten, die an einen Port geschickt werden, besitzen eine identische Struktur und lassen sich anhand eines Feldes in Ereignisklassen unterteilen. Ein gutes Beispiel sind die IDCMP Nachrichten; anhand des Feldes **Class** werden die Nachrichten in die verschiedenen Nachrichtenklassen unterteilt (z.B. **IDCMP_CLOSEWINDOW**, **IDCMP_RAWKEY** etc). Was liegt also näher, als wiederum Handler für jede Nachrichtenklasse einzuführen?

Die Klasse **PortC** beschreibt die Nachrichtenports, die Nachrichten empfangen können. Die Methoden **SignalHandlerC::forMe** und **SignalHandlerC::handle** sind in dieser Klasse geeignet überladen: Die Methode **PortC::forMe** liefert **TRUE**, sobald eine Nachricht am Nachrichtenport bereitliegt (beachtet also die Signalmaske nicht) und die Methode **PortC::handle** holt alle anliegenden Nachrichten vom Port und übergibt sie nacheinander der Methode **PortC::handleMsg**. Diese Methode muß die Nachricht verarbeiten und u.U. an den Sender zurückschicken.

Die Klasse **HandlerPortC** baut auf der Klasse **PortC** auf und erweitert diese um das Handlerkonzept für Nachrichten: Aufbauend auf der Klasse **MessageHandlerC** werden einem Nachrichtenport dieser Klasse Nachrichtenhandler übergeben (**HandlerPortC::add**).

Erhält ein Nachrichtenport der Klasse **HandlerPortC** eine Nachricht, so wird jeder Nachrichtenhandler gefragt, ob er diese Nachricht brauchen kann (**MessageHandlerC::forMe**). Falls dem so ist, wird die Methode **MessageHandlerC::handle** des Handlers aufgerufen, die diese Nachricht verarbeitet:

Das Verlassen der Methode **SignalsC::loop** wird hier im Unterschied zu den Signalhandlern nicht durch das Ergebnis der Behandlungsmethode der Nachricht ausgelöst, sondern durch eine eigene Methode **MessageHandlerC::exit**, die aufgerufen wird, wenn die Behandlungsmethode **MessageHandlerC::handle** das Ergebnis **TRUE** liefert.

Die Fensterklasse WindowC

Die Fensterklasse `WindowC` enthält die Fenster der Intuition Library. Diese Fenster können in einem offenen und geschlossenen Zustand existieren (im Gegensatz zu Intuition, wo man die Fensterstruktur erst durch das Öffnen des Fensters erhält).

```
#include <classes/intuition/window.h>
#include <classes/intuition/screen.h>
#include <classes/exec/libraries.h>

class TestWindowC : public WindowC {
public:
   TestWindowC(IDCMPortC &, STRPTR title);
   ~TestWindowC();
private:
   WindowCloseHandlerC wch;
};
```

Aussehen und Verhalten der Fenster wird mit den gleichen Tags festgelegt wie unter Intuition (also alle Tags die mit `WA_` beginnen). Eine Ausnahme stellt das Tag `WA_Gadgets` dar, die Gadgetverwaltung geschieht über Gadgetlisten, für deren Anbindung an das Fenster einige Methoden zur Verfügung stehen. Deshalb darf dieses Gadget nicht angegeben werden (bzw. wird unbarmherzig aus der Tagliste gefiltert).

```
TestWindowC::TestWindowC(IDCMPortC &p, STRPTR title)
   : WindowC(p,
      WA_DragBar,TRUE,
      WA_Width,300,
      WA_Height,150,
      WA_CloseGadget,TRUE,
      WA_DepthGadget,TRUE,
      WA_Title,title,
      WA_IDCMP,IDCMP_GADGETUP,
      TAG_END),
```

Die Tags können sowohl im Konstruktor, wie auch in der Methode `WindowC::open` angegeben werden

- Tags, die in der Open-Methode verwendet werden, haben nur für diesen einen Open-Befehl Wirkung.
- Tags, die im Konstruktor angegeben werden, haben für jeden Open-Befehl Wirkung.
- Einige Methoden verändern die Tagliste, die im Konstruktor angegeben wurde; damit können Methoden, die auch Sinn bei geschlossenem Fenster machen, auch dann verwendet werden. Dazu zählt das Setzen des Fenstertitels, des Bildschirmtitels und die Größenveränderung des Fensters.

Die IDCMP-Flags können sowohl über die Tags, als auch über die entsprechenden Methoden `WindowC::setIDCMP`, `WindowC::addIDCMP`, `WindowC::subIDCMP` gesetzt werden.

Zur Verarbeitung der Fenster-spezifischen IDCMP Nachrichten steht eine Klasse **WindowEventHandlerC** zur Verfügung. Objekte dieser Klasse verarbeiten Nachrichten einer bestimmten IDCMP Ereignisklasse, die zu einem bestimmten Fenster gehören. Dieses Fenster wird im Konstruktor des Handlers angegeben. Der Konstruktor macht jedoch noch mehr: Er addiert das entsprechende IDCMP- Flag zum Fenster und hängt sich selbst in die Liste der Handler des IDCMP Nachrichtenports des Fensters ein.

```
    wch( *this)
{
```

Für Handler, die nicht an ein bestimmtes Fenster gebunden sind, z.B. die Handler **GadgetUpHandlerC** und **GadgetDownHandlerC** zur Verarbeitung der Ereignisklasse **IDCMP_GADGETUP** und **IDCMP_GADGETDOWN** müssen die IDCMP-Flags extra im Fenster eingetragen werden.

```
  addIDCMP(IDCMP_GADGETUP|IDCMP_GADGETDOWN);
}

TestWindowC::~TestWindowC()
{
  close();
}
```

Fenster können "busy" gesetzt werden; dann wird der Mauszeiger für das Fenster auf die Uhr umgesetzt (unter V39 wird der Voreinsteller beachtet) und das Fenster nimmt keinerlei Eingaben mehr an. In diesem Beispiel wird dieser Status über die Fensterliste gesetzt (s.u.).

Die Fensterklasse **WindowC** stellt zwei Methoden zur Verfügung, um den Zustand des Fensters auch über die Lebensdauer eines Fensterobjekts zu speichern. Die Klasse **WindowSnapshotC** stellt Objekte zur Verfügung, die die Position und Größe sowie den Busy-Zustand des Fenstesr speichern können. Mit der Methode **snapshot** schreibt das Fenster seinen Zustand auf ein Objekt der Klasse **WindowSnapshotC**, mit der Methode **stamp** übernimmt ein Fenster diesen Zustand aus einem solchen Zustand. Die Daten des Snapshot-Objekts sind einfache Datentypen und können daher einfach in einer Voreinstellungsdatei gespeichert werden, um den Fensterzustand auch über den Programmlauf hinweg zu retten.

Hier folgt nun noch das Hauptprogramm, um die oben definierte Fensterklasse `TestWindowC` zu verwenden:

```
LibraryBaseErrC GadToolsBase("gadtools.library",37);
LibraryBaseErrC UtilityBase("utility.library",37);
LibraryBaseErrC CxBase("commodities.library",37);
LibraryBaseErrC LayersBase("layers.library",37);
LibraryBaseErrC WorkbenchBase("workbench.library",37);

int main()
{
  if (!LibraryBaseC::areAllOpen())
    return 20;

  // diese Klasse empfängt und verarbeitet alle Signale
  SignalsC sc;
```

IDCMP Nachrichten für alle Fenster können über einen einzigen Nachrichtenport empfangen werden. Die Klasse `IDCMPortC` kann die IDCMP Nachrichten empfangen und durch spezielle IDCMP Ereignishandler verarbeiten. Ein Beispiel ist die Klasse `WindowCloseHandlerC`, deren Objekte das Ereignis `IDCMP_CLOSEWINDOW` verarbeiten (und im Normalfall die Applikation beenden). Die Klasse `TestWindowC` verwendet diesen Handler.

```
  IDCMPortC port;
  sc.add(port);

  // einige Fenster
  TestWindowC window1(port, "Fenster 1: busy");
  TestWindowC window2(port, "Fenster2");
  TestWindowC window3(port, "Fenster 3: busy");
```

Objekte der Fensterklasse `WindowC` können in einer Liste verwaltet werden. Die Klasse `WindowListC` stellt dazu geeignete Objekte zur Verfügung. Fenster, die in einer solchen Liste eingetragen sind, können alle auf einmal geschlossen werden, oder alle, bis auf eine eventuelle Ausnahme, in den "busy" Zustand versetzt werden. Dadurch lassen sich auf einfache Weise modale Dialoge realisieren (im allgemeinen sollte man jedoch versuchen, modale Dialoge zu vermeiden).

```
  WindowListC windowliste;
  windowliste.add(window1);
  windowliste.add(window2);
  windowliste.add(window3);

  // alle Fenster öffnen
  window1.open(WA_Left,0,WA_Top,0,TAG_END);
  window2.open(WA_Left,20,WA_Top,20,TAG_END);
  window3.open(WA_Left,40,WA_Top,40,TAG_END);

  windowliste.setBusy(TRUE,&window2);

  sc.loop();
```

```
    windowliste.close();

    return 0;
}
```

Die Fensterklasse GTWindowC

Die Klasse **GTWindowC** unterscheidet sich äußerlich nicht von der Klasse **WindowC**, der Unterschied findet sich in der Eigenart der GadTools Library, die eine geringfügig andere Implementation des Fensteröffnens und -refreshs verlangt. Ebenso erwarten Fenster der Klasse **GTWindowC** einen IDCMP Nachrichtport der Klasse **GTIDCMPortC**, da auch der Nachrichtenempfang durch die GadTools-Library eine andere Implementation bedingt.

Gadgets und Gadgetlisten

Die Klassenbibliothek unterstützt die beiden gebräuchlichen Arten der Gadgets: BOOPSI Gadgets und GadTools Gadgets. Die Klasse **GadgetC** bildet dazu eine Basis für alle Gadgets und definiert damit ein technisches Modell für alle Gadgets.

Dieses technische Modell erlaubt eine generelle Behandlung der Position und Größe des Gadgets, Auswahl des Zeichensatzes für Textausgaben des Gadgets und einige Methoden zur Verwaltung in Gadgetlisten.

Ein abstraktes Gadget ist die Gadgetliste. Die Klasse **GadgetListC** erbt von der Klasse **GadgetC**, dadurch können Bäume von Gadgets gebildet werden. Die Gadgetliste nimmt weitere Gadgets der Klasse **GadgetC** auf und erlaubt, die Anwendung verschiedener Methoden auf alle Gadgets der Liste (z.B. den Zeichensatz zu ändern).

Neben diesem technischen Modell existiert noch das logische Modell der verschiedenen Gadgettypen. Jedes Gadget zeigt ein bestimmtes Verhalten, das unabhängig davon ist, ob es ein GadTools oder ein BOOPSI Gadget ist, oder wie das Verhalten konkret in Aussehen und Bedienung umgesetzt ist. Z.B. enthält die Klassenbibliothek ein BOOPSI-Cycle-Gadget, das die Auswahl über ein Popup-Menü erlaubt. Obwohl sich darin die Bedienung für den Benutzer sichtlich vom Cycle-Gadget der GadTools-Library unterscheidet, bleibt die Bedienung der Gadgets durch den Programmierer identisch. Also sind die beiden Gadgets (das Cycle-Gadget der GadTools-Library und das neue BOOPSI-Gadget) von dem gleichen logischen Modell abgeleitet: der Klasse **GCycleC**.

Jedes Gadget der GadTools-Library und viele der neuen BOOPSI-Gadgets erben also nicht nur von der Klasse **GadgetC**, sondern auch von einer der logischen Modellklassen (die alle von **GModelC** erben).

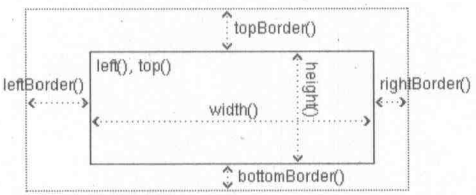
Neben diesen beiden Klassen, die die Basis eines jeden Gadgets bilden, verwenden die Gadgets ein Objekt der Klasse **GadgetEventC**, das die Ereignisbehandlung des Gadgets festlegt. Ein Gadget kann verschiedene Ereignisse auslösen, z.B. ein **IDCMP_GADGETUP** Ereignis, oder - über einen speziellen Handler - ein **IDCMP_RAWKEY** Ereignis. Diese Ereignisse werden von passenden Handlern

aufgefangen. Die Handler rufen dann die zu dem Ereignis passende Methode der Klasse **GadgetEventC** auf, z.B. die Methode **GadgetEventC::up** im Falle eines **IDCMP_GADGETUP** Ereignisses.

Die Fensterklasse **WindowC** stellt einige Methoden zur Verfügung, mit denen eine Gadgetliste und ein Fenster assoziiert werden. Gadgets dürfen in ein Fenster der Klasse **WindowC** nur über eine solche Fensterliste eingetragen werden, das Tag **WA_Gadgets** ist nicht erlaubt (und wird durch die Methode **WindowC::open** ausgefiltert).

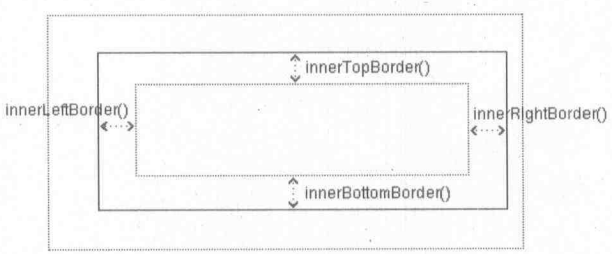
Der Layouter

Der Layouter positioniert Rechtecke auf einer rechteckigen Grundfläche. Dabei wird sowohl die Position wie die Größe der Rechtecke berechnet. Jedes Rechteck legt darüberhinaus einen Rahmen fest, der das Rechteck umgibt. Die Breite dieses Rahmens, der natürlich an allen vier Seiten des Rechtecks unterschiedlich breit sein kann, wird durch das Rechteck bestimmt, nicht durch den Layouter! Die folgende Abbildung zeigt ein Rechteck mit Rahmen und einige Gadgets mit unterschiedlichen Rahmen, die durch die Position und Größe der Beschriftungen festgelegt sind.



Das Modell eines solchen Rechtecks für den Layouter bildet die Klasse **`LayoutC`**. Sie enthält Methoden zur Positionierung und Größenveränderung des Rechtecks und der Ermittlung der Rahmengröße.

Neben diesem Grundmodell kennt der Layouter noch eine erweitere Form: das Gruppenrechteck. Diese Form steht für Rechtecke, in denen selbst wieder weitere Rechtecke liegen können. Dieses Rechteck kennt neben der Position, Größe und dem äußeren Rahmen auch noch einen inneren Rahmen. Dieser Rahmen bestimmt den Abstand der innenliegenden Rechtecke zum Rand. Die folgende Abbildung zeigt ein Gruppenrechteck mit äußeren und inneren Rahmen.



Das Modell eines Gruppenrechtecks für den Layouter bildet die Klasse **`GroupLayoutC`**, die von der Klasse **`LayoutC`** erbt. Sie enthält zusätzliche Methoden zur Ermittlung der inneren Rahmenbreite und zur Bestimmung einer Minimalgröße des Rechtecks.

Diese beiden Modelle werden durch den Layouter verwendet, um die Elemente (Gadgets) zu positionieren und in der Größe zu verändern. Um zum Beispiel einen Button zu erhalten, der durch den Layouter verwendet werden kann, muß man die Klasse des Buttons (z.B. **`BIButtonC`**) zusammen mit der Klasse **`LayoutC`** zu einer Klasse „layoutbarer" Buttons vererben (z.B. **`LBIButtonC`**). Die

abstrakten Methoden der Klasse `LayoutC` sind dann passend mit den Methoden zur Positionierung und Größenveränderung der Buttonklasse zu überladen.

Die „layoutbaren" Gadgets akzeptieren alle einen besonderen Wert für die Breite und Höhe des Gadgets: `LAYOUT_AUTOSIZE`. Wenn die Größe der Gadgets festgelegt wird (durch die Methoden `LayoutC::layoutWidth` und `LayoutC::layoutHeight`), erkennen die Gadgets, wenn sie mit dieser speziellen Größenangabe konstruiert wurden. Daraufhin berechnen sie die nötige Größe für sich selbst automatisch.

Die Buttonklasse enthält nun Objekte, die durch den Layouter verarbeitet werden können. Allerdings werden damit noch keine Regeln festgelegt, die das Layout beschreiben, d.h. die Position und Größe des Buttons in Abhängigkeit der anderen Elemente auch tatsächlich festlegen.

Diese Regeln werden durch Objekte der Klassen `GeometryC` und `GeometryGroupC` festgelegt. Zu jedem Objekt der Klasse `LayoutC` gehört ein Objekt der Klasse `GeometryC`, und zu jedem Objekt der Klasse `GroupLayoutC` ein Objekt der Klasse `GeometryGroupC`.

Durch die Trennung der Elemente, die durch den Layouter verarbeitet werden von der Geometrie eines Elements können die Regeln des Layouts von den in diesem Layout verwendeten Elementen unabhängig entwickelt werden. Zum Beispiel sind dadurch Bausteine für das Layout denkbar, die immer wieder eingesetzt werden können.

Wie wird nun eine Geometrie eines Rechtecks festgelegt? Im Prinzip ganz einfach: Ein Rechteck hat 4 Koordinaten (links, rechts, oben, unten) und entsprechend wird für jede Koordinate eine Regel angegeben. Diese Regel bestimmt in irgendeiner Art den Wert füt diese Koordinate, z.B. kann dadurch die Koordinate sich auf die Koordinate eines anderen Elements beziehen, oder auf den Rand des Gruppenrechtecks, in dem das Rechteck liegt.

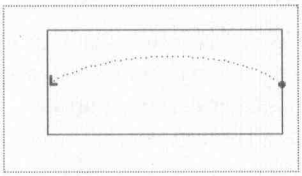
Die Regeln zur Bescheibung eines Layouts

Die Regeln werden im Konstruktor der Klassen `GeometryC` und `GeometryGroupC` angegeben. Jede Regel für jede der 4 Koordinaten besteht aus 3 Teilen:

- Einer der 14 Bezeichner für die Regelart.
- Ein Pointer auf ein Objekt der Klasse `GeometryBaseC`, üblicherweise ein Objekt der Klasse `GeometryC` oder `GeometryGroupC` oder `NULL`.
- Eine Zahl, die entweder einen Pixeloffset oder eine Prozentzahl angibt; da der Pixeloffset immer addiert wird, ist er in Regeln für die rechte oder untere Koordinate normalerweise negativ.

Die 14 Bezeichner bedeuten im einzelnen:

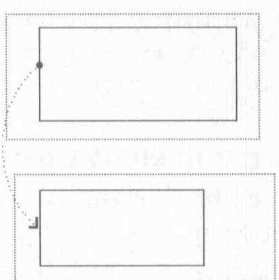
LAYOUT_SIZE



Jedes Gadget kennt eine eigene Größe. Diese kann entweder bei der Initialisierung des Gadgets angegeben, oder durch das Gadget selbst berechnet werden (typischerweise durch Angabe des speziellen Werts `LAYOUT_AUTOSIZE` für die Breite oder Höhe des Gadgets).

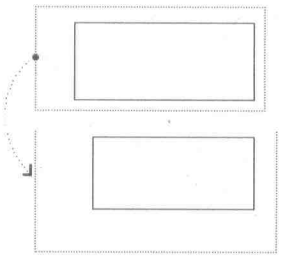
Diese Regel bedeutet, daß die entsprechende Koordinate sich im Abstand der Größe des Gadgets auf die Position der gegenüberliegenden Koordinate bezieht. Wenn Sie also für die linke Koordinate diese Regel angeben, so wird damit das Gadget so breit wie das Gadget eben ist, wobei die rechte Koordinate in einem gültigen Layout die Position festlegen muß. Deshalb können nicht beide Koordinaten nach dieser Regel berechnet werden.
Der Pointer muß `NULL` und der Offset `0` sein.

LAYOUT_SAME



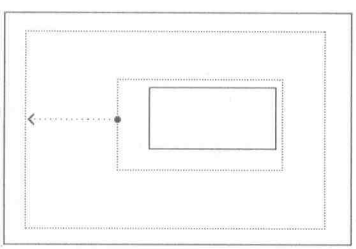
Diese Regel legt die Koordinate auf den gleichen Wert der gleichen Koordinate eines anderen Rechtecks. Zu dieser Regel müssen Sie die Geometrie des anderen Rechtecks angeben und eine Distanz, die zusätzlich zu der Koordinate addiert wird.

LAYOUT_SAMEBORDER



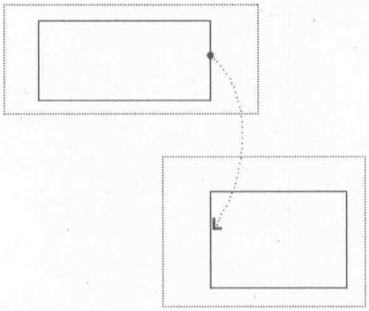
Diese Regel ist im Prinzip der oberen gleich, beachtet jedoch zusätzlich den Rand der beiden Rechtecke. Durch diese Regel liegen die Ränder beider Rechtecke auf der gleichen Position, nicht die Koordinate des Rechtecks selbst. Wiederum kann eine zusätzliche Distanz angegeben werden.

LAYOUT_GROUP



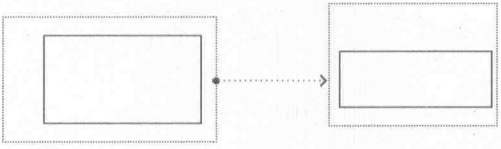
Diese Regel bezieht den Rand des Rechtecks auf den inneren Rand des umgebenden Rechtecks. Eine sehr häufig gebrauchte Regel. Es kann eine Distanz zum Rand angegeben werden.

LAYOUT_OPP



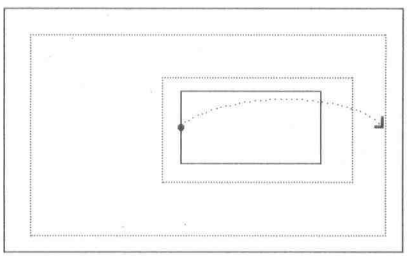
Diese Regel entspricht der Regel **LAYOUT_SAME**, bezieht sich allerdings auf die gegenüberliegende Koordinate des anderen Rechtecks. Es kann eine zusätzliche Distanz zwischen den Objekten angegeben werden.

LAYOUT_OPPBORDER



Diese Regel entspricht der Regel **LAYOUT_SAMEBORDER**, bezieht sich allerdings auf die gegenüberliegende Koordinate des anderen Rechtecks. Damit lassen sich sehr einfach Rechtecke in eine Zeile nebeneinander oder in eine Spalte untereinander legen. Es kann eine zusätzliche Distanz zwischen den Objekten angegeben werden.

LAYOUT_OPPGROUP



Diese Regel entspricht der Regel `LAYOUT_GROUP`, bezieht sich aber auf die gegenüberliegende Seite der umgebenden Gruppe. Da nur die Distanz dann noch einen Abstand zum Rand schaffen kann, ist das eine eher selten gebrauchte Regel.

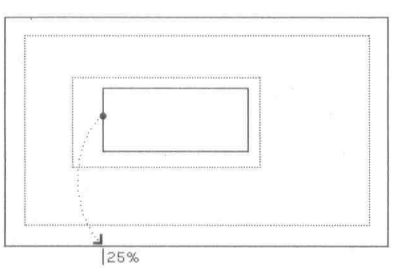
LAYOUT_MAXSIZE

Diese Regel ermittelt die Größe aller Rechtecke der Gruppe, und setzt die Koordinate in diesem Abstand zur Position der gegenüberliegenden Koordinate. Der Offset kann einen Minimalwert für die Größe angeben.

LAYOUT_SAMESIZE

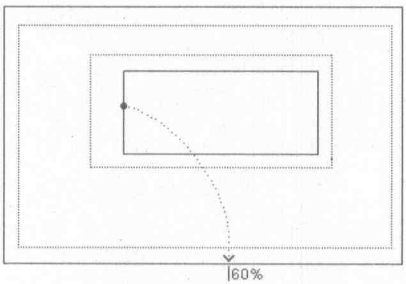
Diese Regel ermittelt die Größe des anderen Rechtecks, nachdem dieses bearbeitet wurde, und setzt die Koordinate in diesem Abstand zur Position der gegenüberliegenden Koordinate. Der Offset muß 0 sein.

LAYOUT_PROCENT



Diese Regel ermittelt die Breite des umgebenden Rechtecks und setzt die Koordinate auf einen Prozentteil dieser Breite. Der Prozentteil muß statt einer Distanz angegeben werden.

LAYOUT_PROCENTCENTER



Diese Regel ermittelt die Breite des umgebenden Rechtecks und des eigenen Rechtecks. Dann setzt sie die Koordinate so, daß die Mitte des Rechtecks auf dem angegebenen Prozentteil der Breite des umgebenden Rechtecks zu liegen kommt.

LAYOUT_MAXSAMEBORDER

Diese Regel ermittelt die maximale Breite des entsprechenden Randes aller Rechtecke und setzt die Koordinate in diesem Abstand zur Position der gegenüberliegenden Koordinate.

LAYOUT_MAXOPPBORDER

Diese Regel entspricht der oberen Regel, berechnet allerdings die maximale Größe des gegenüberliegenden Randes aller Rechtecke.

LAYOUT_HOOK

Obwohl das nun ein ganzer Haufen Regeln sind, könnte es ja noch nicht ausreichen. Tatsächlich gibt es z.B. ein Gadget, mit dem das Layout durch den Benutzer manuell verändert werden kann. Um solche besonderen Regeln zu implementieren, dient diese Regel. Die Verwendung dieser Regel ist allerdings noch privat (d.h. nicht dokumentiert).

Durch Kombination dieser Regeln ist so gut wie jedes "schöne" Layout erstellbar. Alle Regeln beachten die Einteilung der Geometrien in die Gruppen. Wenn also die beiden Prozentregeln den Prozentanteil an der Breite oder Höhe ermitteln, so wird als Berechnungsgrundlage die Breite oder Höhe der Gruppe verwendet, in der die Geometrie, die die Prozentregel benutzt, liegt. Gleiches gilt auch für die Regeln, die die maximale Größe oder Rahmenbreite ermitteln.

Dennoch können mutige Layouter über Gruppengrenzen hinweg Elemente ansprechen. Es ist allerdings nicht gesagt, ob das immer funktioniert. Im wesentlichen kommt das auf die exakte Abhängig-

keit der Elemente untereinander an. Im Zweifel einfach ausprobieren. Übrigens: Jedes Layout, daß mit einem Zeichensatz korrekt aussieht, sollte es auch mit jedem anderen tun - insofern Sie keine Pixelangaben für die Größen der Elemente verwendet haben.

Der Layoutvorgang

Der Layouter enthält eine Klasse `RootGeometryC`. Sie bildet die Wurzel des Geometrienbaums.

Diese Klasse enthält eine wichtige Methode: `RootGeometryC::layout`. Damit wird das Layout in ein Rechteck, dessen linke, obere Ecke und Größe angegeben wird, eingefügt.

Diese Methode akzeptiert den speziellen Wert `LAYOUT_AUTOSIZE` für die Größe des Rechtecks. Dann wird die minimal benötigte Größe für das Layout berechnet. Beim ersten Layoutvorgang oder nach wesentlichen Änderungen des Layout (z.B. Änderungen am Geometrienbaum, oder Änderungen an den Gadgets, die die Ränder größe oder automatische Größe der Gadgets beeinflussen (Fontänderung!)) muß zuerst ein Layout mit dieser speziellen Größenangabe berechnet werden.

Damit tatsächlich die minimale Größe berechnet wird, sollten Sie für alle Gadgets, deren Breite oder Höhe durch den Layoutvorgang bestimmt wird, und nicht durch das Gadget selbst (also z.B. linke und rechte Koordinate haben die Regel `LAYOUT_GROUP`) für die entsprechende Größenangabe den Wert `LAYOUT_AUTOSIZE` verwenden. Tun Sie dies nicht, erkennt das Gadget bei wiederholten Layoutvorgängen nicht, daß es seine eigene Größe automatisch berechnen soll und nimmt deshalb die vom Layouter berechnete. Diese kann allerdings inzwischen (z.B. durch Veränderung der Größe des Fensters, in dem das Layout liegt) sehr groß geworden sein - der Layoutvorgang würde also nicht die minimale Größe, sondern eine sehr viel größere berechnen.

Eine Knopfzeile

Dieses Beispiel zeigt den Aufbau eines Bausteins für den Layouter: Häufig stehen am unteren Rand eines Requesters für Voreinstellungen drei Knöpfe: **"Speichern"**, **"Benutzen"**, **"Abbrechen"**. Dieser Baustein positioniert diese 3 Knöpfe in einer Zeile:

```
class ThreeButtonGroupC : public GeometryGroupC {
public:
```

Dem Konstruktor wird ein Objekt für den Rahmen, drei Knöpfe und die Regeln, wie dieser Baustein ins Layout eingepasst wird, übergeben.

```
    ThreeButtonGroupC(
      GroupLayoutC &frame,
      LGButtonC &left,
      LGButtonC &middle, LGButtonC &right,
      ULONG topRelation, GeometryC *topObject, WORD topOffset,
      ULONG bottomRelation, GeometryC *bottomObject, WORD bottomOffset,
      ULONG leftRelation, GeometryC *leftObject, WORD leftOffset,
      ULONG rightRelation, GeometryC *rightObject, WORD rightOffset);
private:
  GeometryC leftGeo;
```

```
    GeometryC middleGeo;
    GeometryC rightGeo;
};
```

Der Konstruktor macht nun nichts mehr, als die Gruppe aufzubauen:

```
ThreeButtonGroupC::ThreeButtonGroupC(
      GroupLayoutC &frame,
      LGButtonC &left,
      LGButtonC &middle, LGButtonC &right,
      ULONG topRelation, GeometryC *topObject, WORD topOffset,
      ULONG bottomRelation, GeometryC *bottomObject, WORD bottomOffset,
      ULONG leftRelation, GeometryC *leftObject, WORD leftOffset,
      ULONG rightRelation, GeometryC *rightObject, WORD rightOffset)
    : GeometryGroupC(frame,
      topRelation, topObject, topOffset,
      bottomRelation, bottomObject, bottomOffset,
      leftRelation, leftObject, leftOffset,
      rightRelation, rightObject, rightOffset),
```

Der linke Knopf wird um die 20 Prozentmarke zentriert, erhält die Maximalgröße der 3 Knöpfe, aber mindestens 40 Pixel breit - dadurch wirken Knöpfe mit sehr kurzen Aufschriften ("Yes", "No", "OK") angenehmer.

```
      leftGeo(left,
      LAYOUT_SIZE,NULL,0,LAYOUT_GROUP,NULL,0,
      LAYOUT_PROCENTCENTER,NULL,20,LAYOUT_MAXSIZE,NULL,40),
```

Der mittlere Knopf wird um die 50 Prozentmarke zentriert. Beachten Sie die Angabe der Geometrie des linken Knopfes in der Prozentregel: Diese wird zur Größenberechnung des Platzbedarfs dieser Gruppe benötigt; die Regel schafft dann mindestens soviel Platz, daß die beiden Knöpfe sauber nebeneinander liegen.

```
      middleGeo(middle,
      LAYOUT_SIZE,NULL,0,LAYOUT_GROUP,NULL,0,
      LAYOUT_PROCENTCENTER,&leftGeo,50,LAYOUT_MAXSIZE,NULL,0),
```

Der rechte Knopf wird symmetrisch zum linken um die 80 Prozentmarke zentriert. Auch hier wird wiederum der linke Nachbar, also der mittlere Knopf in der Prozentregel zur Platzberechnung angegeben.

```
      rightGeo(right,
      LAYOUT_SIZE,NULL,0,LAYOUT_GROUP,NULL,0,
      LAYOUT_PROCENTCENTER,&middleGeo,80,LAYOUT_MAXSIZE,NULL,0)
{
    GeometryGroupC::add(leftGeo);
    GeometryGroupC::add(middleGeo);
    GeometryGroupC::add(rightGeo);
}
```

Dieser Baustein ist nun universell für solche 3-Knopf-Zeilen verwendbar. Manchmal ist er vielleicht zu universell. Man könnte z.B. eine neue Gruppe ableiten, die gleichzeitig von einer Gadgetliste erbt, und die Gadgets für die Knöpfe und den Rahmen der Gruppe darin festlegen und zu dieser Liste addieren. Wie allgemein oder speziell Sie ihre Gruppen aufbauen, kommt auf den Kontext an, indem Sie die neuen Gruppen und Gadgetlisten verwenden wollen.

Eine Gadgetspalte

Dieses Beispiel zeigt eine universell einsetzbare Gruppe, die eine beliebige Anzahl von Geometrien in einer Spalte anordnet. Die Elemente haben dabei alle den gleichen Abstand zum linken und rechten Rand der Gruppe.

```
class GadgetColumnC : public GeometryGroupC {
public:
   GadgetColumnC(
      GroupLayoutC &frame,
      ULONG topRelation, GeometryC *topObject, WORD topOffset,
      ULONG bottomRelation, GeometryC *bottomObject, WORD bottomOffset,
      ULONG leftRelation, GeometryC *leftObject, WORD leftOffset,
      ULONG rightRelation, GeometryC *rightObject, WORD rightOffset);
```

Zu der Gruppe werden die gewünschten Geometrien addiert. Diese Geometrien können mit beliebigen Regeln erzeugt sein, da die Methode `add` die Regeln geeignet überschreibt (s.u.)

```
   VOID add(GeometryC &);
private:
```

Die transparenten Geometrien werden benötigt, um den Platz für den rechten und linken Rand zu berechnen.

```
   TransparentLayoutC left, right;
   GeometryC leftG, rightG;
   GeometryC *lastG;
};

GadgetColumnC::GadgetColumnC(
      GroupLayoutC &frame,
      ULONG topRelation, GeometryC *topObject, WORD topOffset,
      ULONG bottomRelation, GeometryC *bottomObject,
      WORD bottomOffset,
      ULONG leftRelation, GeometryC *leftObject, WORD leftOffset,
      ULONG rightRelation, GeometryC *rightObject, WORD rightOffset)
   : GeometryGroupC(frame,
      topRelation,topObject,topOffset,
      bottomRelation,bottomObject,bottomOffset,
      leftRelation,leftObject,leftOffset,
      rightRelation,rightObject,rightOffset),
      left(),
      right(),
```

Zur Berechnung des linken Randes wird ein transparentes Element verwendet, dessen Geometrie links an die Gruppe gehängt wird und durch die Regel der rechten Koordinate eben so groß gemacht wird, wie das Maximum der linken Ränder alle Geometrien in der Gruppe.

```
leftG(left,
LAYOUT_SIZE,NULL,0,LAYOUT_GROUP,NULL,0,
LAYOUT_GROUP,NULL,0,LAYOUT_MAXOPPBORDER,NULL,0),
```

Für den rechten Rand wird entsprechend die Geometrie an den linken Gruppenrand gehängt und das Element so breit gemacht, wie die maximale Breite für die rechten Ränder angibt.

```
rightG(right,
LAYOUT_SIZE,NULL,0,LAYOUT_GROUP,NULL,0,
LAYOUT_MAXOPPBORDER,NULL,0,LAYOUT_GROUP,NULL,0),
lastG(NULL)
{
  GeometryGroupC::add(leftG);
  GeometryGroupC::add(rightG);
}

VOID GadgetColumnC::add(GeometryC &g)
{
```

Falls schon ein Element in die Liste eingehängt wurde, wird das neue Element an den unteren Rand dieses Elements angehängt.

```
  if (lastG)
    g.setTopRule(LAYOUT_OPPBORDER,lastG,2)
  else
```

Das erste Element wird allerdings an den oberen Rand der Gruppe angehängt.

```
    g.setTopRule(LAYOUT_GROUP,NULL,0);
```

Jedes Element ist so hoch, wie es sein Element angibt.

```
  g.setBottomRule(LAYOUT_SIZE,NULL,0);
```

Das Element wird links und rechts an die jeweiligen transparenten Elemente angehängt. Daß für den Rand des Elements genügend Platz ist, garantieren die Geometrien der transparenten Elemente durch die Verwendung der Regel **LAYOUT_MAXOPPBORDER**.

```
  g.setLeftRule(LAYOUT_OPP,&leftG,0);
  g.setRightRule(LAYOUT_OPP,&rightG,0);
  GeometryGroupC::add(g);
  lastG = &g;
}
```

Diese Gruppe kann nun allgemein eingesetzt werden, eine entsprechende Gruppe für Zeilen von Gadgets zu erstellen sollte kein Problem sein. Es müssen nur die entsprechenden Regeln vertauscht werden. In der Demo-Schublade liegt ein Beispiel für die Gadgetspalte bereit.

Die Grenzen des Layouters

Der Layouter kann nicht jede theoretisch denkbare Kombination aller Regeln korrekt berechnen - eine Implementationseinschränkung betrifft die beiden Prozentregeln:

- Wenn die Größe der Fläche, in die das Layout passen soll, bekannt ist, d.h. in der Layoutanweisung angegeben wird (z.B. eine vorgegebene Größe eines Fensters), so funktionieren diese Regeln einwandfrei. Soll jedoch die notwendige Größe der Fläche erst berechnet werden (und das geschieht in der Klasse `LayouterWindowC` vor dem Öffnen des Fensters), so kommen die Prozentregeln an ihre Grenzen, denn eine Prozentangabe gibt nunmal keine notwendige Größe der umgebenden Fläche vor.

Die Prozentregeln versuchen dennoch ihr bestes:

- Die umgebende Fläche wird wenigstens so groß gemacht, daß das Element mit der Prozentregel ganz hineinpasst. Wenn also dieses Element direkten Kontakt mit der umgebenden Gruppe hat (durch die Regel `LAYOUT_GROUP` in der anderen Koordinate), so kann die notwendige Fläche berechnet werden. Ebenso kann man bei den Prozentregeln ein Element angeben, neben dem das Element mit der Prozentregel ohne Überlappung liegen soll - das obige Beispiel "Eine Knopfzeile" verwendet diese Möglichkeit, um genügend Platz in der Zeile zu erhalten.

Es ist also sinnvoll, die Prozentregeln immer von einem Rand ausgehend anzugeben (d.h. die Elemente mit der Prozentregel sollten entweder direkt Kontakt mit dem Gruppenrand oder einem Element mit einer Prozentregel haben); um dies zu erreichen muß man unter Umständen die Elemente in geeignete Gruppen zusammenfassen.

Die anderen Regeln haben diese Schwierigkeiten in der Bestimmung des benötigten Platzes nicht.

Zur Gestaltung der Oberfläche

Der Autor will hier natürlich keine neuen Richtlinien für das Aussehen oder gar Verhalten der Oberfläche geben. Allerdings wird man feststellen, das man die Oberflächen, mit den Elementen, die durch diese Bibliothek zur Verfügung stehen, auf verschiedene Arten graphisch gestalten kann.

Erstens der Einsatz von Rahmen zur graphischen Gruppierung der Elemente. Dazu steht eine Klasse `LBBevelboxC` zur Verfügung, die Rahmen mit Titel zeichnet. Ab Version 39 des Betriebssystems sind dies normalerweise in die Oberfläche eingeritzte Rahmen , unter älteren Version die einfachen eingesenkten oder hervorstehenden Rahmen.

Die zweite Möglichkeit verwendet Patterns, zur farblichen Gruppierung der Elemente. Dies wird ebenfalls durch die Klasse `LBBevelboxC` unterstützt, hierbei sollte man jedoch als Rahmenart den

einfachen eingesenkten Rahmen verwenden (der einzige Rahmen der unter Version 37 und 38 zur Verfügung steht).

Die dritte Möglichkeit ist die Gruppierung durch horizontale und vertikale Striche, die die Klasse `LBBevellineC` zur Verfügung stellt.

Selbstverständlich können diese Möglichkeiten auch kombiniert werden (in manchen Fällen läßt sich das auch kaum vermeiden), jedoch sollte man damit sparsam umgehen - der Benutzer könnte durch eine zu musterreiche und zu stark unterteilte Oberfläche eher verwirrt werden, als dadurch Orientierung zu erhalten.

Die Fensterklasse LayouterWindowC

Die Klasse `LayouterWindowC` erbt von der Klasse `GTWindowC` und stellt Fenster zur Verfügung, deren Gadgets durch den Layouter verwaltet werden. Die Größe des Fensters wird beim Öffnen automatisch berechnet und kann durch den Benutzer verändert werden. Im Konstruktor wird ein Zeichensatz angegeben, der für alle Gadgets verwendet wird, er kann jederzeit verändert werden.

Die Fensterfläche ist in 5 Gebiete unterteilt: die innere Fläche (in der die meisten Gadgets liegen) und die 4 Ränder des Fensters.

In den unteren und rechten Rand werden häufig Rollgadgets gelegt, um den sichtbaren Teilbereich eines dargestellten Dokuments zu verschieben. Die folgende Klasse zeigt, wie diese Gadgets in den Rand gelegt werden:

```
class ScrollerWindowC : public LayouterWindowC {
public:
   ScrollerWindowC(GTIDCMPortC &, ScreenC &);
   ~ScrollerWindowC();
   TextAttrC font;
   LBScrollerC vertical,horizontal;
   GeometryC verticalGeo,horizontalGeo;
};

ScrollerWindowC::ScrollerWindowC(GTIDCMPortC &p, ScreenC &s)
   : LayouterWindowC(p,s,font,
     WA_SizeBRight,TRUE,
     WA_SizeBBottom,TRUE,
     WA_CloseGadget,TRUE,
     WA_DepthGadget,TRUE,
     WA_SizeGadget,TRUE,
     WA_DragBar,TRUE,
     TAG_END),
     font("topaz.font",8),
     vertical(NULL,*this,0,LAYOUT_AUTOSIZE,
      GA_RightBorder,TRUE,
      PGA_Borderless,TRUE,
      PGA_Freedom,FREEVERT,
      PGA_NewLook,TRUE,
      TAG_END),
```

```
        horizontal(NULL,*this,LAYOUT_AUTOSIZE,0,
         GA_BottomBorder,TRUE,
         PGA_Borderless,TRUE,
         PGA_Freedom,FREEHORIZ,
         PGA_NewLook,TRUE,
         TAG_END),
        verticalGeo(vertical,
         LAYOUT_GROUP,NULL,0,LAYOUT_GROUP,NULL,-2,
         LAYOUT_GROUP,NULL,2,LAYOUT_GROUP,NULL,-2),
        horizontalGeo(horizontal,
         LAYOUT_GROUP,NULL,1,LAYOUT_GROUP,NULL,-1,
         LAYOUT_GROUP,NULL,0,LAYOUT_GROUP,NULL,-2)
{
```

Die Gadgets werden in 3 Listen gespeichert: Titel-, BOOPSI- und GadTools-Gadgets. Die BOOPSI- und GadTools-Gadgets erhalten immer den gleichen Zeichensatz, die Titel-Gadgets (die natürlich normalerweise auch BOOPSI-Gadgets sind) sind zur graphischen Gruppierung der anderen Gadgets gedacht und können einen eigenen Zeichensatz erhalten. Die 3 Gadgetlisten heißen **gadgets**, **gtgadgets** und **titlegadgets**.

```
    gadgets.add(vertical);
    gadgets.add(horizontal);
```

Die 5 Geometriegruppen heißen **innerGeo** für die innere Fläche; **topGeo**, **bottomGeo**, **leftGeo** und **rightGeo** für die 4 Ränder.

```
    rightGeo.add(verticalGeo);
    bottomGeo.add(horizontalGeo);
}

ScrollerWindowC::~ScrollerWindowC()
{
  close();
}
```

Die Zeichensätze der Gadgets werden als Pointer angegeben, und müssen deshalb gültig sein, solange das Fenster geöffnet ist.

Die Fensterklasse StandardWindowC

Die Klasse **StandardWindowC** erbt von der Klasse **LayouterWindowC** und vereinfacht deren Gebrauch. Die Angabe der Zeichensätze geschieht zwar weiterhin über Pointer auf Objekte der Klasse **TextAttrC**, jedoch werden diese Zeichensatzbeschreibungen kopiert.

Außerdem kann im Konstruktor der Klasse **StandardWindowC** für ein Fenster ein Default-Gadget angegeben werden. Dieses Gadget erhält einen eigenen Zeichensatz, der immer **fett** eingestellt ist,

aber in Größe und Familie mit dem normalen Zeichensatz übereinstimmt. Dieser fette Zeichensatz kann auch noch für andere Zwecke verwendet werden.

Beim ersten Öffnen des Fensters wird automatisch der Standardzeichensatz des Bildschirms, auf dem das Fenster geöffnet wird, für beide Zeichensätze des Fenster verwendet.

Datei- und Verzeichnissperren

Wenn man auf den Verzeichniseintrag einer Datei, und nicht deren Inhalt zugreifen möchte, z.B. um zu testen, ob eine bestimmte Datei existiert, sollte man Dateisperren (Locks) verwenden.

Die Klasse `FileLockC` ermöglicht den einfachen Umgang mit den Dateisperren. Es stehen in dieser Klasse alle Funktionen der DOS-Library, die auf Dateisperren operieren, zur Verfügung. Einige Methoden benötigen weitere Strukturen der DOS-Library, z.B. die Methode `FileLockC::examine` benötigt einen `FileInfoBlock`, auf dem sie bestimmte Informationen über eine Datei oder ein Verzeichnis ablegt. Diese Methoden existieren immer in zwei Varianten:

- Eine Variante erwartet einen Pointer auf die Struktur der DOS-Library, eine andere Variante erwartet eine Referenz auf ein Objekt einer entsprechenden Klasse (z.B. `FileInfoBlockC` zur Struktur `FileInfoBlock`), die die Verwaltung dieser Struktur für sie übernimmt (insbesondere Allozierung und Freigabe).

Die Klasse `FileLockC` hat einen Erben: die Klasse `DirLockC`. Diese Klasse erweitert die Klasse `FileLockC` um Methoden, die typisch für Operationen auf Verzeichnissen sind, insbesondere natürlich das Einrichten eines neuen Verzeichnisses (während neue Dateien durch Objekte der Klasse `FileHandleC` erzeugt werden können). Desweiteren enthält diese Klasse Methoden zum „Scannen" eines Verzeichnisses.

Bis auf das Programm "*cd*" darf kein Programm das aktuelle Verzeichnis des CLI ändern. Wenn Sie dennoch innerhalb ihres Programms das aktuelle Verzeichnis ändern wollen, müssen Sie es am Ende des Programms unbedingt wieder auf den Anfangswert zurücksetzen. Die Klasse `CurrentDirC` erleichtert Ihnen diese Arbeit, indem bei der Konstruktion eines Objekts dieser Klasse das Objekt das aktuelle Verzeichnis speichert, und bei der Destruktion das aktuelle Verzeichnis wieder auf das Gespeicherte zurücksetzt.

Wenn Sie also vorhaben, in ihrem Programm das aktuelle Verzeichnis zu ändern, sollten Sie einfach am Anfang der `main()`-Funktion oder `wbmain()`-Funktion ein Objekt dieser Klasse deklarieren (eine globale Variable des Typs `CurrentDirC` bringt ebenfalls den gewünschten Effekt).

Die Dateiklassen

Die eigentliche Dateiklasse `FileHandleC` der Bibliothek basiert direkt auf den Funktionen der DOS-Library.

Die Lese- und Schreibmethoden (`FileHandleC::read` und `FileHandleC::write`) operieren wie die Originalfunktionen `Read()` und `Write()` auf typenlosen Puffern. Hier stellt jedoch die Klassenbibliothek einen geeigneten dynamischen Puffer zur Verfügung, den Sie zum Lesen und Schreiben einer Datei verwenden können. Die Klasse `BufferC`.

Sie können die Puffergröße eines Objekts der Klasse `BufferC` jederzeit ändern, müssen jedoch beachten, daß sich dadurch natürlich die Adresse des Puffers ändert. Sie dürfen daher die Adresse nicht in einer Variablen zwischenspeichern, sondern sollten diese Adresse bei jeder Verwendung des Puffers neu ermitteln.

Das Kopieren einer Datei könnte dann ungefähr so aussehen:

```
FileHandleC von, nach;
if (von.open(VON_NAME, MODE_OLDFILE) && nach.open(NACH_NAME, MODE_NEWFILE))
{
   BufferC puffer(10000);
   ULONG i;
   while ((i = von.read(puffer.buffer(),puffer.size())) > 0)
   {
      if (nach.write(puffer.puffer(),i) < 0)
        break;
   };
};
```

Die Dateien werden natürlich im Destruktor der Klasse `FileHandleC` automatisch geschlossen.

Von der Klasse `FileHandleC` erben zwei weitere Klassen: die Klasse `InputHandleC` für die Standardeingabe und die Klasse `OutputHandleC` für die Standardausgabe. Im Normalfall ist es jedoch sinnvoller, die Objekte `cin` und `cout` aus der Header-Datei *<iostream.h>* für diesen Zweck zu verwenden, nur wenn Sie globale Variablen in Ihrem Programm komplett vermeiden wollen, sollten Sie auf die Klassen `InputHandleC` und `OutputHandleC` zurückgreifen.

Kommandozeilenargumente

Die Klassenbibliothek vereinfacht die Auswertung der Kommandozeilenargumente stark. Sie müssen nur eine Variable des Typs `ArgsC` deklarieren, dem Konstruktor dieser Klasse übergeben Sie das Template der Argumente.

Danach können Sie die verschiedenen Argumentenwerte direkt ermitteln, entweder durch Angabe des Argumentennamens, den Sie im Template angegeben haben, oder der Nummer des Arguments. Die Klasse `ArgsC` unterscheidet dabei String-Argumente, Zahl-Argumente und Schalter-Argumente.

Ebenso einfach ist es, Mehrfachargumente auszuwerten: Die Klasse `ArgsMCursorC` stellt Objekte zur Verfügung, deren Konstruktor Sie wiederum die Nummer oder den Namen des Mehrfacharguments übergeben müssen. Dann können Sie auf die Werte des Arguments durch die typischen Methoden eines Cursors zugreifen.

Beschreibung des HotHelp Projekts zu EASY-OBJECTS

Das Handbuch zu EASY-OBJECTS beschreibt die Grundkonzepte und Verwendung der Klassenbibliothek, für die täglich Arbeit steht Ihnen eine ausführliche Hypertextdokumentation zur Verfügung. Das HotHelp-Projekt EASY-OBJECTS, das bei der Installation der Klassenbibliothek ebenfalls installiert wird.

Bevor Sie die folgende Beschreibung des HotHelp-Projekts EASY-OBJECTS lesen, sollten Sie sich mit der Bedienung von HotHelp vertraut gemacht haben.

Öffnen Sie das HotHelp-Fenster (üblicherweise durch den Tastendruck Alt-Help) und wählen Sie die Projektliste aus (Knopf "Projekte"). Es erscheint die Startseite des Projekts EASY-OBJECTS.

Unter der Titelgraphik befinden sich die Referenzen des Inhaltsverzeichnisses:

Einführung

Diese Referenz führt Sie zu einem Beispiel, wie EASY-OBJECTS benutzt wird; es wird eine besondere Version des unvermeidlichen Programms "HelloWorld" vorgestellt. Dieses Kapitel befindet sich auch im Handbuch.

Hintergrund

Diese Referenz führt Sie zu einem weiteren Inhaltsverzeichnis, in dem Sie die restlichen Kapitel des Handbuchs (bis auf diese Beschreibung natürlich) finden. Sie haben also die freie Wahl, ob Sie die Verwendung der Bibliothek lieber am Monitor, oder aus dem Handbuch lernen wollen...

Diese Kapitel enthalten ausführlich dokumentierte Beispiele, die den jeweiligen konzeptionellen oder technischen Aspekt der Klassenbibliothek vorstellen. Die Beispiele liegen selbstverständlich auch als lauffähige Programme in der Demo-Schublade von MaxonC++ vor.

Technische Kapitel

Dieser Überschrift folgt einer Referenzenliste, jede Referenz verweist auf ein Inhaltsverzeichnis eines Teilbereichs der Klassenbibliothek. Die Einteilung folgt der Einteilung der Betriebssystemdokumentation, z.B. Exec, DOS, Intuition etc. In diesen Kapiteln finden Sie die Beschreibung der Klassen, die Ressourcen aus den entsprechenden Bibliotheken des Betriebssystems verwalten, also Klassen zur Speicherverwaltung, zu Signalen und Nachrichtenports in Exec, Klassen zu Dateien und Locks in DOS und Klassen zur Fensterverwaltung und Gadgets in Intuition.

Drei Kapitel fallen dabei aus der Reihe:

Das Kapitel Boopsi

ist aus Intuition ausgegliedert und beschreibt die Klassen des Boopsi-Systems. Darin befinden sich sowohl die BOOPSI-Klassen des Betriebssystems, wie auch neue BOOPSI-Gadgets und Images.

Das Kapitel DataStructures

beschreibt Klassen, die verschiedene Datenstrukturen zur Verfügung stellen: dynamische Arrays, dynamische Datenpuffer, Stringverwaltung usw.

Das Kapitel Layouter

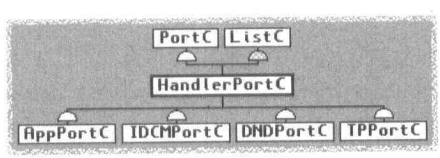
beschreibt die Klassen des Layouters, mit dem Sie zeichensatzunabhängige und größenveränderbare Bedienoberflächen erzeugen können.

Die Namen aller Kapitel stimmen übrigens mit dem Namen des Verzeichnisses überein, in dem die Header-Dateien des jeweiligen Kapitels liegen.

Alle technischen Kapitel besitzen einen identischen Aufbau:

Das Inhaltsverzeichnis des Kapitels führt sämtliche Klassennamen dieses Teilbereichs als Referenzen auf. Jede Referenz führt zu der Beschreibung einer Klasse.

Zuerst wird kurz die Bedeutung der Klasse beschrieben, darauf folgt eine Graphik, in der Sie erkennen können, welche Klassen diese Klasse beerbt, und an welche wichtigen Klassen diese Klasse vererbt.



Die Abbildung zeigt den Klassenbaum der Klasse **HandlerPortC**, der Name dieser Klasse steht in der mittleren Zeile und ist dick umrandet. Darüber stehen die beiden Klassen, die **HandlerPortC** beerbt, **PortC** und **ListC**. In der unteren Zeile stehen zwei Klassen, an die **HandlerPortC** erbt, **AppPortC** und **IDCMPortC**. Es folgt der Name der Header-Datei, in der die Definition der Klasse steht.

Wenn Sie eine Header-Datei einbinden wollen, so sollten Sie aus Geschwindigkeitsgründen die Include-Anweisung mit einem `#ifdef` ... `#endif` klammern. Das Symbol, auf dessen Existenz Sie in der `#ifdef`-Anweisung testen, setzt sich dabei aus dem Namen der Header-Datei zusammen.

Alle Header-Dateien liegen im Verzeichnis "*Classes*" im Include-Pfad des Compilers, entsprechend beginnt das Symbol mit"`CPP_`". Der zweite Teil des Symbols ist der großgeschriebene Name des Verzeichnisses in dem die Header-Datei liegt, der dritte Teile ist der Name der Datei, wobei das Suffix "*.h*" durch "*_H*" ersetzt und außerdem alles groß geschrieben wird. Verständlicher als viele Worte sind wohl einige Beispiele:

Header-Datei: Classes/Exec/Lists.h ---> `CPP_EXEC_LISTS_H`

Header-Datei: Classes/Layouter/Windows.h ---> `CPP_LAYOUTER_WINDOWS_H`

Eine vollständige Include-Anweisung sollte dann folgendermaßen aussehen:

```
#ifndef CPP_EXEC_LISTS_H
#include <classes/exec/lists.h>
#endif
```

Als nächstes folgt die C++ Definition der Klasse. In dieser Defintion sind natürlich auch alle anderen Klassennamen referenziert, insbesondere können Sie darüber auch die Beschreibung zu den Elternklassen lesen.

Ein Tip

HotHelp kann Textteile exportieren, z.B. ins Clipboard legen oder direkt in den Editor einfügen. Wenn Sie eine Klasse ableiten, und wollen Methoden überladen, können Sie die benötigten Prototypen direkt aus dem HotHelp Projekt in Ihren C++ Quelltext exportieren.

Dieser Klassendefintion folgt eine Zeile in der alle abgeleiteten Klassen (nicht nur die wichtigen Klassen aus der Graphik) aufgeführt sind. Damit sind Sie in der Lage, den gesamten Klassenbaum abzulaufen. Gute Dienste leistet Ihnen dabei der Knopf "***Vortext***" von HotHelp, durch den Sie auf einen Vorgängertext zurückblättern.

Der Rest jeder Seite enthält die Beschreibungen aller wichtigen Konstruktoren und Methoden. Diese werden mit dem Prototyp und einer kurzen Beschreibung dargestellt.

Jede Methode besitzt einen unsichtbaren Schlüssel, der sich aus dem qualifizierten Methodennamen zusammensetzt (z.B. `PortC::handleMsg`, also erst der Klassenname, dann zwei Doppelpunkte und schließlich der Methodenname). In der Beschreibung tauchen immer wieder Referenzen auf diese unsichtbaren Schlüssel auf, die Sie unmittelbar zur Beschreibung einer Methode führen - z.B. wenn ein Methode einer Klasse eine Methode einer Elternklasse überlädt.

Benutzung des HotHelp Projekts EASY-OBJECTS

Die Klassenbibliothek enthält über 300 verschiedene Klassen. Selbstverständlich können Sie nicht alle Klassen auswendig kennen, darum werden Sie ständig auf das HotHelp Projekt zurückgreifen müssen, um die Namen der Klassen oder auch Prototypen der Methoden nachzuschlagen.

Hier folgt nun ein Beispiel, wie Sie beim Aufbau eines Programms Hilfe durch das HotHelp Projekt bekommen.

Das Programm soll ein Fenster mit einem Knopf darin öffnen. Wenn man den Knopf drückt, soll auf der Standardausgabe der Text **"Knopf gedrückt"** ausgegeben werden. Das Programm soll entweder durch das Schließsymbol des Fensters oder durch <Ctrl>+<C> beendet werden können.

Wenn Sie nicht wissen, wie die geeignete Fensterklasse überhaupt heißt, dann öffnen Sie das HotHelp Fenster und gehen Sie in das Projekt **"EASY-OBJECTS"**. Für Bedienoberflächen sollte man immer den Layouter verwenden, also gehen Sie in das Kapitel **"Layouter"**. Dort finden sich zwei Einträge zu Fenstern: **LayouterWindowC** und **StandardWindowC**. Lesen Sie sich beide Beschreibungen der Klassen durch, dann werden Sie feststellen, daß die Klasse **StandardWindowC** die geeignete Klasse ist.

Also leiten wir eine Klasse von dieser Klasse ab:

```
class KnopfWindowC : public StandardWindowC {
```

Der Konstruktor muß natürlich die nötigen Parameter enthalten, die auch die Elternklasse benötigt.

```
public:
   KnopfWindowC(GTIDCMPortC &, ScreenC &);
   ~KnopfWindowC();
```

Nun wollen wir einen Knopf für den Layouter verwenden. Auch hier können wir wieder in HotHelp im Kapitel **"Layouter"** nachschauen. Dort finden sich einige Gadgets, die Knöpfe sind: ***LBButtonC***, ***LBFrButtonC***, ***LBIButtonC*** und ***LGTButtonC***. Die Namenskonvention von EASY-OBJECTS ist dabei folgende: das "*L*" am Anfang der Namen steht für den Einsatz der Klasse im Layouter, das folgende "*B*" für BOOPSI-Gadgets, oder das folgende "*GT*" für GadTools-Gadgets. Die beiden Klassen `LBButtonC` und `LBFrButtonC` sind, wie Sie feststellen können, wenn Sie die Klassen bis zu ihren Elternklassen `BButtonC` und `BFrButtonC` verfolgen, die C++ Klassen zu den System BOOPSI Klassen `BUTTONGCLASS` und `FRBUTTONCLASS`. Beide sind nicht sonderlich einfach zu verwenden, die Klasse `LBIButtonC` ist ein BOOPSI-Knopf, der deutlich besser den Layouter unterstützt. Wählen wir also diesen Knopf (der GadTools-Knopf ginge genauso).

Anhand des Konstruktors sehen wir, daß ein Knopf der Klasse `LBIButtonC` wie jedes andere Gadget auch, eine Instanz der Klasse `GadgetEventC` benötigt, die die Reaktion auf die Betätigung durch den Benutzer festlegt. Dafür deklarieren wir eine Klasse `KnopfEventC`, die wir später noch genauer bestimmen.

Zudem benötigt jedes Element im Layouter eine Geometrie, diese deklarieren wir auch noch.

```
private:
   KnopfEventC knopfEvent;
   LBIButtonC knopf;
   GeometryC knopfGeo;
```

Um das Programm durch das Schließsymbol des Fensters zu beenden, benötigen wir einen geeigneten Handler, der die IDCMP Nachricht verarbeitet. Im Kapitel Intuition findet sich bei den Fensterklassen eine Klasse namens `WindowCloseHandlerC` - klingt eigentlich ganz brauchbar. Die Beschreibung dieser Klasse bzw. der Methode `WindowCloseHandlerC::close()` erfüllt genau unsere Wünsche.

```
   WindowCloseHandlerC wch;
};
```

Die Implementation des Konstruktors ergibt sich automatisch aus den Konstruktoren der ausgewählten Klassen. Denken Sie daran, daß Sie mit Shift-Help direkt das Wort unter dem Cursor in HotHelp suchen können. Stellen Sie den Cursor einfach auf den Namen einer verwendeten Klasse, um den Prototyp des Konstruktors oder einer Methode nachzulesen.

```
KnopfWindowC::KnopfWindowC(GTIDCMPortC &p, ScreenC &s)
   : StandardWindowC(p,s,NULL,
     TAG_END),
```

Um zur Beschreibung der BOOPSI-Klasse für den Knopf zu kommen, muß man 2 Referenzen durchlaufen: Die Elternklasse `BIButtonC` der Klasse `LBIButtonC` nennt den Namen der BOOPSI-Klasse: `IButtonClass`. Dort finden sich alle neuen Attribute dieser Klasse. Für unseren Zweck reichen jedoch die Standardattribute `GA_Text` (das immer für Beschriftungstexte verwendet wird) und `GA_RelVerify` (das immer auf `TRUE` gesetzt werden muß, damit Intuition die Betätigung des Gadgets an das Programm weitergibt).

```
     knopf(&knopfEvent,*this,LAYOUT_AUTOSIZE,LAYOUT_AUTOSIZE,
      GA_Text,"Ein Knopf",
      GA_RelVerify,TRUE,
      TAG_END),
```

Die Beschreibung der Regeln, die Sie im Konstruktor der Geometrie angeben, finden Sie auch in HotHelp unter dem Schlüssel **"Die Regeln"**.

```
     knopfGeo(knopf,
      LAYOUT_GROUP,NULL,2,LAYOUT_GROUP,NULL,-2,
      LAYOUT_GROUP,NULL,2,LAYOUT_GROUP,NULL,-2),
     wch(*this)
{  gadgets.add(knopf);
   innerGeo.add(knopfGeo);
}
```

```
KnopfWindowC::~KnopfWindowC()
{
  close();
}
```

Die Klasse `KnopfEventC`, die wir im Fenster verwenden, muß noch definiert werden. Sie wird von der Klasse `GadgetEventC` abgeleitet (dies konnten wir an der Defintion des Konstruktors der Gadgetklasse `LBIButtonC` sehen).

```
class KnopfEventC : public GadgetEventC {
public:
  KnopfEventC() : GadgetEventC() { };
```

Diese Methode wird aufgerufen, wenn das Gadget gedrückt und wieder losgelassen wurde. Den Prototyp können Sie einfach aus HotHelp exportieren. Da nur eine einfache Ausgabe vorgenommen wird, ist die Methode inline deklariert.

```
  VOID up(WindowC *, GadgetC *, IntuiMessageC * )
    { cout << "Knopf gedrückt.\n"; };
};
```

Das Hauptprogramm soll hier nicht mehr im Einzelnen beschrieben werden, es sieht in allen Programmen gleich aus.

Natürlich müssen die verschiedenen Header-Dateien der Klassen, die im Programm benutzt werden, geladen werden. Welche Dateien Sie benötigen, können Sie auch in HotHelp nachlesen, der Name steht über der jeweiligen Klassendefinition.

Hier folgt nun der vollständige (und richtig sortierte) Quelltext des Beispiels, das natürlich auch in der Demo-Schublade vorliegt.

```
#include <classes/Layouter/Windows.h>
#include <classes/Layouter/BoopsiGadgets.h>
#include <classes/Exec/Libraries.h>
#include <iostream.h>

class KnopfEventC : public GadgetEventC {
public:
  KnopfEventC() : GadgetEventC() { };
  VOID up(WindowC *, GadgetC *, IntuiMessageC * )
    { cout << "Knopf gedrückt.\n"; };
};

class KnopfWindowC : public StandardWindowC {
public:
  KnopfWindowC(GTIDCMPortC &, ScreenC &);
  ~KnopfWindowC();
private:
  KnopfEventC knopfEvent;
  LBIButtonC knopf;
  GeometryC knopfGeo;
  WindowCloseHandlerC wch;
```

```
};

KnopfWindowC::KnopfWindowC(GTIDCMPortC &p, ScreenC &s)
   : StandardWindowC(p,s,NULL,
     TAG_END),
     knopf(&knopfEvent,*this,LAYOUT_AUTOSIZE,LAYOUT_AUTOSIZE,
      GA_Text,"Ein Knopf",
      GA_RelVerify,TRUE,
      TAG_END),
     knopfGeo(knopf,
      LAYOUT_GROUP,NULL,2,LAYOUT_GROUP,NULL,-2,
      LAYOUT_GROUP,NULL,2,LAYOUT_GROUP,NULL,-2),
     wch(*this)
{
   gadgets.add(knopf);
   innerGeo.add(knopfGeo);
}

KnopfWindowC::~KnopfWindowC()
{
   close();
}

LibraryBaseErrC GadToolsBase("gadtools.library",37);
LibraryBaseErrC UtilityBase("utility.library",37);
LibraryBaseErrC CxBase("commodities.library",37);
LibraryBaseErrC LayersBase("layers.library",37);
LibraryBaseErrC WorkbenchBase("workbench.library",37);

int main()
{
   if (!LibraryBaseC::areAllOpen())
     return 20;

   SignalsC sc;

   PublicScreenC screen();
   if (!screen.lock(NULL))
     return 100;

   GTIDCMPortC port;
   sc.add(port);
   GadgetUpHandlerC uphandler;
   port.add(uphandler);
   KnopfWindowC window(port,screen);
   CtrlCHandlerC ctrlchandler;
   sc.add(ctrlchandler);
   window.open();
   sc.loop();

   return 0;
}
```

AMIGA

MaxonC++

Source-Level-Debugger

MAXON
computer

1. Einführung

Das Entwicklungssystem MaxonC++ 3 enthält einen völlig neu entwickelten Quellsprachen-Debugger. Der bisher vertriebene Debugger (des gleichen Names) wird nicht mehr unterstützt. Der Debugger erleichtert die Entwicklung funktionsfähiger Programme wesentlich, da mit ihm Fehler effektiv gesucht und gefunden werden können. Durch seine intuitive Bediendbarkeit und seine konsequente Hochsprachen-Orientierung ist er insbesondere für den Anwender von C++ geeignet.

Einige seiner Eigenschaften:

Mehrere Quelltexte gleichzeitig betrachtbar.

Darstellung aller Datenstrukturen in C bzw. C++ Syntax oder Pascal Syntax, es können sogar gemischte Projekte korrekt debuggt werden.

Läuft asynchron zum untersuchten Programm.

automatische Aktualisierung aller Variablenwerte.

konfigurierbare Oberfläche.

Was können Sie mit MaxonSDB tun?

MaxonSDB eignet sich für die Fehlersuche in Programmen, die mit dem Compiler MaxonC++ (ANSI C oder C++) ab Version 3 oder MaxonPASCAL erstellt sind. Sie können das Programm in einzelnen Schritten oder in größeren Sprüngen abarbeiten, dabei Variablenwerte beobachten und Datenstrukturen inspizieren. Programmierer, deren Rechner eine MMU besitzen, können den Enforcer verwenden um illegale Speicherzugriffe zu testen, mit dem Debugger können Sie direkt die entsprechende Sourcestelle zu einem Enforcerhit suchen. Bislang kann der Debugger keine disassemblierten Programmteile darstellen (bei der Entwicklung wurde wesentlich auf die Hochsprachen Rücksicht genommen) und nicht mehreren Prozesse oder Tasks gleichzeitig beobachten.

Über dieses Handbuch

In diesem Handbuch werden einige Begriffe in einer allgemeineren Bedeutung als üblich verwendet, und für manche Begriffe wird der englische Name genutzt, falls deren deutsche Übersetzung nicht gebräuchlich ist.

Mit **Modul** wird hier eine Übersetzungseinheit bezeichnet, wie sie C oder Assembler-Programmierern bekannt ist. Pascal benutzt hierfür den Begriff „Unit".

Der Begriff **Funktion** deckt sowohl die Funktionen der Sprache C, als auch die Unterprogramme und Prozeduren von Pascal und Assembler ab.

Argumente sind dasselbe wie **Parameter**. Es wird aber unterschieden zwischen den Kommandozeilenargumenten eines Programms und den Argumenten einer Funktion.

Kickstart: Das Betriebssystem der Amiga-Rechner wird oft unter verschiedenen Namen angesprochen, wobei diese Namen eigentlich nur für einen Teil des Betriebssystems stehen (z.B AmigaDOS 2.0 oder Workbench 1.3). In diesem Handbuch wird einheitlich der Name Kickstart für das Betriebssystem des Amiga verwendet. Dabei wird zwischen den Versionen 1.2 und 1.3 nicht unterschieden.

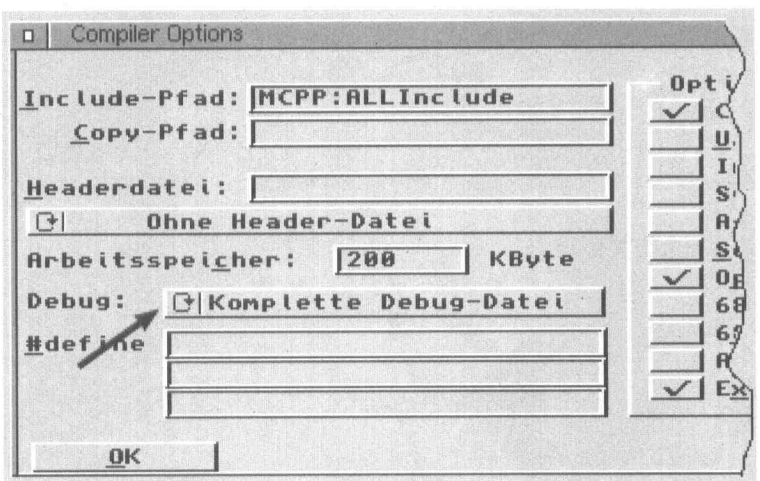
(Der) Debugger.

(Der) Startup-Code. Hierfür gibt es jeweils keinen gebräuchlichen deutschen Begriff.

2. Die Benutzung von MaxonSDB

Die Vorbereitung Ihres Programms

Damit Sie ihr Programm mit MaxonSDB debuggen können, muß ein Schalter in den Compiler-Optionen aktiviert werden. Nach dem Einschalten der Option ist es erforderlich, daß Sie das jeweilige Modul oder alle zum Programm gehörigen Module erneut compilieren! Bei der Kommandozeilenversion des MaxonC^{++} - Compilers muß bei jedem Aufruf die Option „-b" angegeben sein.



Andere Compiler erzeugen eventuell ebenfalls Debug-Dateien, die für MaxonSDB geeignet sind. Bitte entnehmen Sie die dafür nötigen Befehle und Schritte dem Handbuch des jeweiligen Compilers.

Programmstart

Sie können den Debugger auf drei verschiedene Arten starten: von der Workbench, aus der Shell oder direkt aus der Entwicklungsumgebung ihres Compilers.

Der Workbenchstart

Wie gewohnt startet der Doppelklick auf das Programmikon den Debugger. Mit dem Menüpunkt **„öffnen"** aus dem Menü Projekt können Sie die MDBG-Datei des zu debuggenden Programmes laden. Eine weitere Möglichkeit besteht darin, das Programm-Ikon des zu debuggenden Programmes in das Ikonfeld (AppIcon) im Projektfenster abzulegen.

Am einfachsten klicken Sie auf das vom Compiler erzeugte Programm-Ikon mit der Endung "*.mdbg*". Der Debugger wird dann automatisch gestartet und das zu debuggende Programm geladen und ebenfalls gestartet.

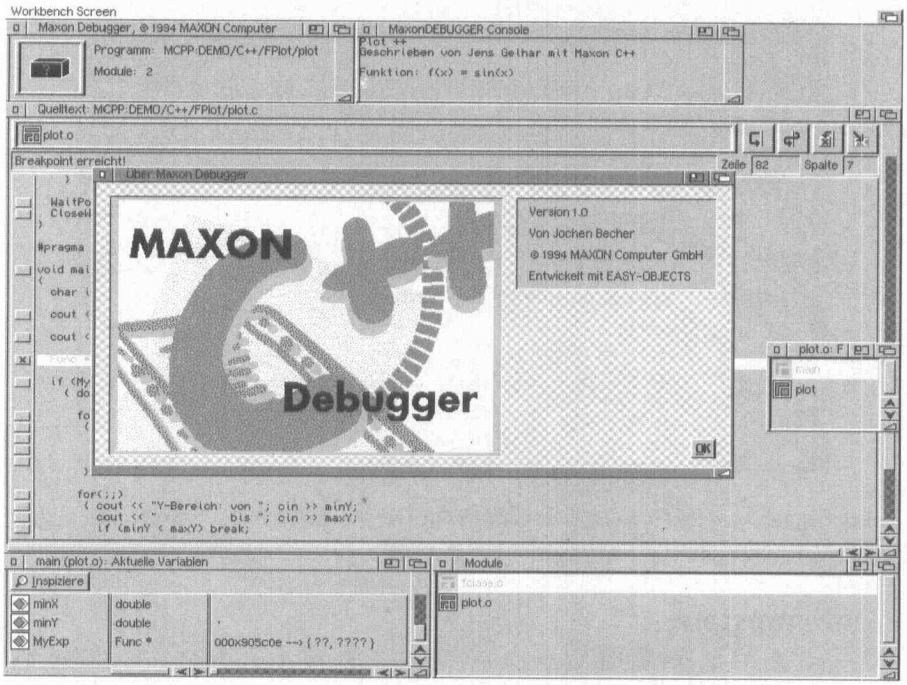
In den Tooltypes von MaxonSDB können folgende Einstellungen vorgenommen werden:

ARGUMENTS="Programmargumente"

Hier müssen die Argumente für das zu debuggende Programm angegeben werden. Die Argumente werden dann bei Programmstart übergeben.

PUBSCREEN=

Dem Tooltype PUBSCREEN können Sie den Namen eines öffentlichen Bildschirms angeben, auf dem alle Debugger-Fenster angezeigt werden sollen. Geben Sie dieses Argument nicht an, wird der Default-Bildschirm verwendet - üblicherweise die Workbench.



Der Shellstart

Starten Sie den Debugger durch Eingabe des Programmnamens MSDB. Als ein Debugger-Argument können Sie den Namen einer MDBG-Datei angeben oder einen Programmnamen. Dieses Programm wird dann in den Debugger geladen und gestartet. Beachten Sie allerdings, daß der Stack wenigstens 40000 Bytes groß sein muß.

Das vollständige Muster der Argumente sieht so aus: PROGRAM, ARGUMENTS, PUBSCREEN/K

Als zweites Argument können Sie die Argumente des zu untersuchenden Programms angeben (benutzen Sie immer Hochkommas für dieses Argument, wenn sie mehrere Argumente an das zu untersuchende Programm übergeben wollen). Das letzte Argument kann der Name eines öffentli-

chen Bildschirms sein, auf dem der Debugger seine Fenster öffnen soll. Geben Sie dieses Argument nicht an, wird der Default-Bildschirm verwendet - üblicherweise die Workbench.

Der Start aus der Entwicklungsumgebung

Dieser Start ist natürlich der einfachste. Kompilieren Sie ihr Programm mit der entsprechenden Compiler-Option um Debugdateien zu erstellen. Wurde Ihr Programm fehlerfrei übersetzt, können Sie den Debugger aus dem Menü **Compiler** starten. Ihr gerade übersetztes Programm wird dann automatisch in den Debugger geladen. Der Debugger übernimmt dabei die Argumente, die Sie im Dialog **Optionen/System** im Textfeld "***Argumente***" angegeben haben.

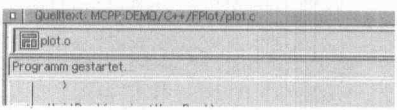
3. Die Fenster des Debuggers

Drag&Drop

In fast jedem Fenster des Debuggers besteht die Möglichkeit, das sogenannte Drag&Drop zu nutzen. Drag&Drop bedeutet ziehen und loslassen, womit schon ungefähr klar wird, was gemeint ist.

Wenn Sie mit der Workbench arbeiten, kennen Sie dieses Prinzip bereits. Es wird beim Kopieren, Verschieben und Installieren genutzt, wenn Sie mit der Maus ein Workbench-Ikon greifen und es über einem anderen Workbenchfenster loslassen.

In MaxonSDB haben viele Fenster Drag&Drop-Bereiche, die Sie daran erkennen können, daß zusätzlich zu einem namentlichen Eintrag ein Ikon angezeigt wird. Beim Klick auf den Namen wird der jeweilige Eintrag ausgewählt. Mit Klick auf das Ikon und Gedrückthalten der Maustaste können Sie das Ikon in den Drag&Drop-Bereich eines anderen Fensters verschieben.

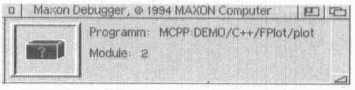


Versuchen Sie es doch gleich einmal. Laden Sie die MDBG-Datei des Programmes FPlot, das bereits mit fertigen Debug-Dateien vorliegt. Öffnen Sie nun ein Fenster für die aktuellen Variablen. Greifen Sie nun im Variablen-Fenster das Ikon des Eintrags `"input"` und ziehen Sie dieses Ikon in den Drag&Drop-Bereich des Quelltextfensters. Sobald Sie den Bereich mit der Maus berühren, wird er invertiert angezeigt und symbolisiert Ihnen so, daß das Ikon hier abgelegt werden kann. Sobald Sie das Ikon über dem invertierten Bereich loslassen wird die Quelltextstelle angezeigt, an der die lokale Variable definiert wird.

Wenn Sie aus dem Funktionen-Fenster Ikons auf den Drag&Drop-Bereich des Quelltextfensters verschieben, können Sie so sehr leicht die Quelltextposition von Funktionen auffinden und zum Beispiel Haltepunkte definieren.

Das Projektfenster

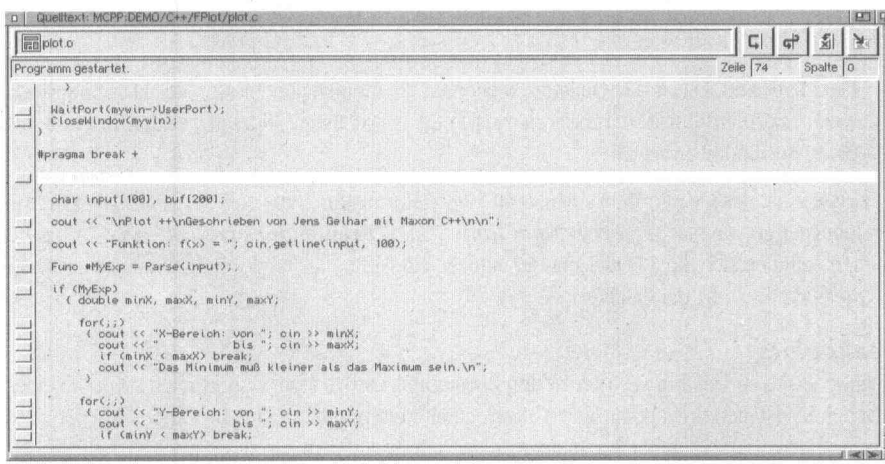
Nach dem Start wird grundsätzlich das Projektfenster des Debuggers geöffnet. Dieses enthält den Namen des geladenen Programms und die Anzahl der Module im Programm. Daneben befindet sich das Ikon des Programms, in dieses Feld können Sie auch ein Ikon der Workbench ziehen, um ein neues Programm zu laden.



Das Projektfenster ist das einzige Fenster, das immer geöffnet ist. Alle anderen Fenster können auch geschlossen sein - je nach Konfiguration ihrer Oberfläche haben Sie von den anderen Fenstern zu Anfang mehr oder weniger geöffnet. Die meisten Fenster können Sie aus dem Menü Fenster öffnen.

Das Quelltextfenster *<A 1>*

Sobald Sie ein Programm geladen und damit auch gestartet haben, wird zumindest ein Quelltextfenster geöffnet, in dem der Quelltext des Moduls mit der Main-Funktion angezeigt wird. Die aktuelle Zeile, d.h. die Zeile in der der PC (Program Counter) des Programms sich gerade befindet, wird invertiert dargestellt.



Das Quelltextfenster ist das wichtigste Fenster. Neben der Anzeige der aktuellen Position des Programms mit der invertierten Programmzeile sind darin auch die Knöpfe enthalten, mit denen der Ablauf des Programms gesteuert wird.

In der obersten Zeile befindet sich der Name des Moduls, dessen Quelltext gerade angezeigt wird. Daneben sind 4 kleine Knöpfe mit Bildsymbolen.

<A I> oder <I>

Mit dem ersten Knopf wird ein Einzelschritt ausgelöst. Das Programm läuft einen Schritt weiter und hält dann wieder an. Bei einem Unterprogrammaufruf wird in das Unterprogramm verzweigt und der entsprechende Quelltext dazu angezeigt.

Der verwendete Compiler bestimmt, wo ein Schritt beginnt und wo er endet. Schritte stimmen dabei mehr oder weniger mit den einzelnen Befehle überein - allerdings werden komplexe Ausdrücke, in denen Funktionsaufrufe stehen, in mehrere Schritte unterteilt.

<A R> oder <R>

Der zweite Knopf führt auch einen Einzelschritt aus, allerdings werden Unterprogramme, die in diesem Schritt aufgerufen werden immer schnell abgearbeitet. Unterprogramme, die nicht Teil des Programms sind (insbesondereBetriebssystemaufrufe) werden sowieso immer schnell abgearbeitet.

<A G> oder <G>

Der dritte Knopf läßt das Programm schnell weiter arbeiten. Dadurch läuft es entweder bis zum Ende durch, oder bis es auf einen Breakpoint stößt.

<A E> oder <E>

Der vierte Knopf setzt den Quelltext wieder auf die momentane Position des Programms, z.B. wenn man den Quelltext verschoben hat, oder sogar einen anderen geladen hat, um z.B. einen Breakpoint zu setzen.

Die zweite Zeile des Quelltextfensters enthält eine Statusmeldung, die darüber Aufschluß gibt, was der Debugger gerade macht (*„Programm läuft"*, oder *"Breakpoint erreicht"* oder ...). Rechts daneben befinden sich die 2 Felder mit der aktuellen Zeile und Spalte des Programms im angezeigten Quelltext. Nun folgt der Quelltext.

Breakpoints

Die linke Spalte im Quelltext-Fenster ist den Breakpoints vorbehalten. Dort können sie durch einen einfachen Klick einen Breakpoint setzen, durch einen weiteren Klick einen Breakpoint löschen. Den gesetzen Breakpoint erkennen Sie an einem kleinen Kreuz auf dem jeweiligen Knopf.

Eine zweite Art von Breakpoints setzen Sie, wenn Sie den Klick auf einen Breakpointknopf bei gedrückter Shifttaste durchführen. Statt einem Kreuz erscheint ein schwarzer Punkt. Dieser Breakpoint wird beim Erreichen automatisch gelöscht.

Den Quelltext können Sie mit den beiden Scrollern sowohl vertikal, wie horizontal verschieben.

Das Modulfenster <A 2>

Das Modulfenster zeigt die Namen aller geladenen Module an. Dieses Fenster besteht aus einer Liste der Modulnamen. Links neben jedem Namen befindet sich ein kleines Ikon, daß Ihnen für Drag & Drop zur Verfügung steht.

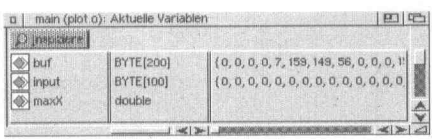
Jedes Programm hat sicher mehrere Module, und die meisten Benutzer (insbesondere die glücklichen Besitzer von Graphikkarten und großen Monitoren) werden auch mehrere Quelltextfenster öffnen. Der einfachste Weg, den Quelltext eines Moduls in einem bestimmten Quelltextfenster anzuzeigen besteht in der Verwendung des Drap&Drop Prinzips: Greifen Sie das Ikon eines Moduls (wie Sie es auch von der Workbench gewohnt sind) und verschieben Sie es auf die erste Zeile eines Quelltextfensters - eben dort wo der Namen des bisher geladenen Modules steht.

Sie werden sehen, daß sich die Fläche dieses Namens invertiert - zum Zeichen, daß dieses Oberflächenelement ein Ikon empfangen kann. Lassen Sie einfach das Ikon los und der Quelltext des Moduls wird geladen.

Das Fenster der globalen Variablen <A 3>

Jedes Modul kennt seine eigenen Variablen. Wählen Sie ein Modul aus, in dem Sie im Modulfenster auf den Namen (nicht auf das Ikon) eines Moduls klicken. Wählen Sie danach aus dem Menü **Fenster** den Punkt **„Globale Variablen"** aus. Es öffnet sich ein Variablenfenster, in dem Sie die globalen Variablen des Moduls sehen. Auch hier gibt es wieder ein paar generelle Worte zu verlieren:

Der Debugger verwendet relativ häufig die Variablenfenster. Diese Fenster bestehen aus drei Listen: Die Liste der Variablennamen, der Typen der Variablen und der momantanen Werte der Variablen.



Die beiden letzten Listen können sehr lang werden - so lang, daß Sie im Fenster nicht passend dargestellt werden können. Deshalb können diese beiden Listen nicht nur vertikal sondern auch horizontal verschoben werden. Die Liste der Variablennamen zeigt auch die aus dem Modulfenster bekannten Ikons - allerdings sehen diese Ikons etwas anders aus, um sie von Modulen unterscheiden zu können. Auch diese Ikons können Sie greifen und woanders fallen lassen - zum Beispiel im Überwachungsfenster, daß Sie ebenfalls aus dem Menü öffnen können. Oder Sie greifen das Ikon und legen es im Quelltextfenster in das Empfangsfeld. Diesmal wird nicht nur das passende Modul geladen, sondern auch noch die Quelltextstelle angezeigt, an der die Variable deklariert ist. Damit können Sie z.B. die exakte Deklaration einer Variablen nachschauen (durch den Preprozessor des C++ Compilers sehen die Deklarationen im Quelltext unter Umständen anders aus, als im Variablenfenster).

Jedes Variablenfenster hat wenigstens einen Knopf über den Listen:

Den Knopf Inspiziere.

Wählen Sie eine Variable aus, indem Sie auf den Namen der Variable klicken. Wenn Sie nun den Knopf Inspiziere drücken, öffnet sich ein neues Variablenfenster, in dem Sie nun die Variable näher betrachten können. Variablen mit einfachen Typen (WORD, ULONG etc) sind nicht sinnvoll zu

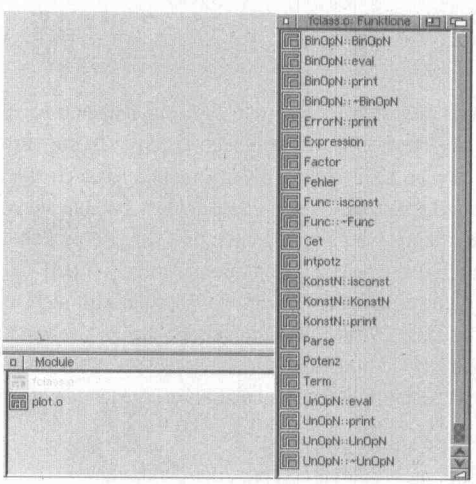
inspizieren, es wird nur der Eintrag aus dem Variablenfenster wiederholt, aber Variablen mit Strukturtypen (struct, union, class, Record) oder Pointertypen und Referenzen können näher betrachtet werden, indem von strukturierten Typen jedes Feld dargestellt wird und Pointer dereferenziert werden.

Von Arrays wird jeder Index einzeln angezeigt. Diese so dargestellten Einträge können im Inspektionsfenster ebenfalls wieder inspiziert werden, z.B. um eine ganze Liste anhand der Pointer zu verfolgen, oder auch Felder von Strukturen näher zu betrachten.

Dabei können Sie zwischen zwei Varianten der Inspektion wählen: Drücken Sie den Inspiziere Knopf mit der Shifttaste zusammen, so wird der bisher dargestellte Text im Inspektionsfenster durch den neuen Inhalt ersetzt. Drücken Sie allerdings den Knopf ohne Shifttaste, wird ein neues Fenster mit dem neuen Inhalt geöffnet. Durch Schließen dieses Fensteres können Sie dann wieder mit der Inspektion an der alten Stelle fortfahren.

Das Fenster der Funktionen <A 4>

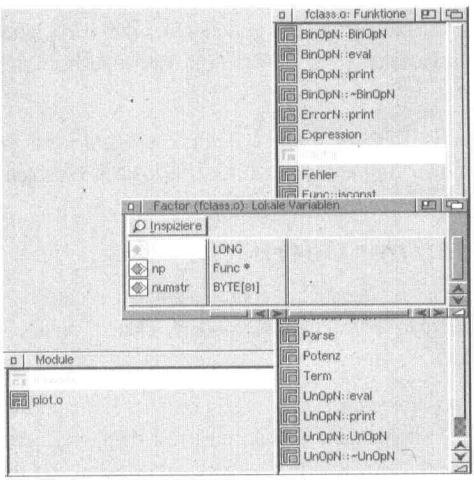
Haben Sie im Modulfenster den Namen eines Moduls ausgewählt, können Sie auch das Fenster der Funktionen dieses Moduls öffnen. Es werden alle Funktionen des Moduls angezeigt - also auch alle Methoden einer Klasse, die in dem Modul deklariert werden. Die Methoden werden dann qualifiziert ausgegeben, d.h. der Klassenname wird mit einem doppelten Doppelpunkt vor den Methodennamen geschrieben.



Die Liste der Funktionen sieht aus, wie die Liste der Module, nur mit demUnterschied, das wiederum das Ikon etwas anders gestaltet ist. Wie im Modulfenster und Variablenfenster können Sie das Ikon greifen und im Quelltextfenster (im Ikonempfangsfeld) ablegen. Wie bei Variablen wird nicht nur das Modul geladen, sondern auch der Quelltext auf den Anfang der Funktionsdeklaration positioniert.

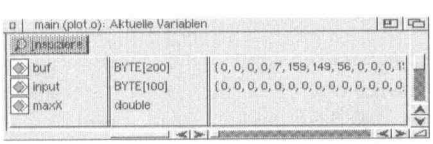
Das Fenster der lokalen Variablen <A 5>

Wählen Sie aus dem Funktionenfenster eine Funktion aus. Jetzt können Sie zu dieser Funktion ein Variablenfenster mit allen lokalen Variablen öffnen, indem Sie den entsprechenden Menüpunkt lokale Variablen aus dem Menü Fenster auswählen. Dieses Fenster ist in Form und Aufbau mit dem der globalen Variablen identisch.



Das Fenster der aktuellen Variablen <A 6>

Öffnen Sie das Fenster der aktuellen Variablen aus dem Menü „**Fenster**" mit dem Menüpunkt **„aktuelle Variablen"**. Dieses Fenster sieht genauso aus wie das Fenster der lokalen Variablen. Allerdings werden in diesem Fenster automatisch immer die Variablen der gerade aktuellen Funktion angezeigt, d.h. der Funktion, in der sich der Programmcounter gerade befindet. Dieses Fenster ist sicher neben dem Quelltextfenster das wichtigste, um sich über den momentanen Zustand des Programms zu informieren.



Das Fenster zur Überwachung <A 7>

Öffnen Sie ein Überwachungsfenster mit dem Menüpunkt **„Überwachung"** aus dem Menü **Fenster**. Sie können von diesen Fenstern beliebig viele öffnen. Dieses Fenster sieht wie die anderen Variablenfenster aus, enthält aber am Anfang keine Einträge. Sie können aber ein Ikon aus einem

beliebigen anderen Variablenfenster greifen und in die Liste der Variablennamen des Überwachungsfensters fallen lassen.

Wie im Quelltextfenster invertiert sich diese Liste, um anzuzeigen, daß diese Liste ein Ikon entgegennehmen kann. Die Variable wird dann im Überwachungsfenster genauso wie in ihrem Ausgangsfenster mit Namen, Typ und Wert dargestellt . Wenn Sie eine Variable nicht länger überwachen wollen, können Sie diese aus der Liste auswählen und mit dem Knopf ***„Entfernen"*** aus dem Fenster entfernen. Dieses Fenster ist besonders dazu geeignet, um den Bildschirm aufzuräumen und nicht unnötig viele Fenster mit globalen (und im Falle von Pascal auch lokalen) Variablen geöffnet zu haben

Die lokalen Variablen sind im Falle von C und C++ am besten im Fenster der aktuellen Variablen zu verfolgen, da man in C und C++ nur jeweils Zugriff auf die lokalen Variablen einer Funktion hat.

Die Konfiguration der Oberfläche

Alle Debugger-Fenster (Projekt-, Quelltext-, Modul-, aktuellen Variablen-, Überwachungs- und das Info-Fenster) können Sie auch beim Programmstart an einer beliebigen Position und mit Ihrer Wunschgröße wieder öffnen.

Öffnen Sie die Fenster, positionieren Sie sie und speichern Sie dann die Konfiguration aus dem Menü **„Konfiguration"**. Sie können die Fenster auch vor dem Speichern wieder schließen, dann wird zwar Position und Größe gespeichert, aber das Fenster beim Start des Debuggers nicht automatisch geöffnet.

Genauso werden auch die Position und Größe aller Fenster gespeichert, die nicht einfach so geöffnet werden können (Filerequester, Fenster der globalen Variablen, der Funktionen, der lokalen Variablen, Inspektionsfenster und Adresseneingabefenster).

Der Filerequester bekommt für alle seine Aufgaben (Laden eines Programms, Konfiguration laden und speichern) die gleiche Position und Größe. Öffnen Sie einfach den Filerequester mit dem Menüpunkt **„Konfiguration laden"**, positionieren Sie ihn und schließen ihn mit Abbruch. Beim nächsten Speichern der Konfiguration werden die Position und Größe dann mitgespeichert. Gleiches gilt auch für das Adreßeingabefenster.

Die übrigen Fenster werden per Beispiel fixiert. Öffnen Sie das erste Mal ein solches Fenster (z.B. des Fenster der globalen Variablen eines Moduls). Positionieren Sie dieses Fenster und schließen es wieder. Beim darauffolgenden Speichern der Konfiguration wird diese Position und Größe als Beispiel gespeichert. Wenn Sie also nach dem nächsten Programmstart des Debuggers ein Fenster mit globalen Variablen öffnen, wird es automatisch an der Position und in der Größe geöffnet, die Sie in ihrem Beispielfenster eingestellt hatten. Das gleiche gilt auch für die Fenster der Funktionen, der lokalen Variablen und der Inspektionsfenster.

Mit dem Menüpunkt **„Konfiguration sichern"** wird die aktuelle Konfiguration in die Datei *„PROGDIR:MSDB.prefs"* gespeichert. Sie können also den Debugger auch aus anderen Verzeichnissen oder aus der Shell starten, ohne ihre Konfiguration zu verlieren.

Mit den Menüpunkten **„Konfiguration sichern als“** und **„Konfiguration laden“** können auch mehrere verschiedene Konfigurationen benutzt werden.

Das Menü Projekt

Der erste Menüpunkt **„öffnen“** <A O> lädt und startet ein Programm. Wenn noch ein Programm läuft wird die Operation mit einer entsprechenden Fehlermeldung abgebrochen. Beenden Sie dann zuerst das laufende Programm.

Der zweite Menüpunkt **„Wieder Starten“** <A W> läßt sich nur ausführen, nachdem ein Programm beendet wurde. Dann wird dieses Programm nochmal geladen und gestartet.

Der Menüpunkt **„Über“** zeigt das Informationsfenster, mit den Copyright-Hinweisen und der Versionsnummer.

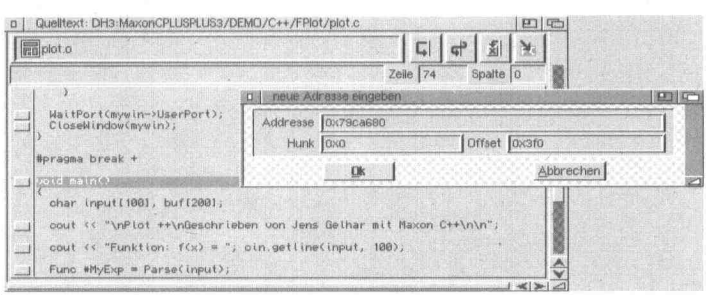
Der letzte Menüpunkt **„Debugger beenden“** <A Q> bricht ebenfalls mit einer Fehlermeldung ab, wenn noch ein Programm läuft. Ansonsten wird der Debugger beendet.

Das Menü Schritt

In diesem Menü finden Sie die Menüpunkte zur Steuerung des Programms. Die ersten vier Menüpunkte stimmen mit den Knöpfen des Quelltextfensters überein:

<A I>	einen Einzelschritt ausführen
<A R>	einen Schritt ausführen und dabei Unterprogramme schnell abarbeiten,
<A G>	bis zum nächsten Breakpoint gehen
<A E>	die aktuelle Position des Programms anzeigen.

Der nächste Menüpunkt **„Neue Adresse zeigen“** läßt sich ebenfalls im Quelltextfenster aufrufen indem Sie den Knopf zur Anzeige des PC (aktuelle Position) mit gedrückter Shifttaste anklicken.



In beiden Fällen öffnet sich das Adreßfenster, in dem Sie entweder eine Hexadezimaladresse oder eine Hunknummer mit Offset eingeben können.

Der Debugger zeigt im Quelltextfenster die passende Quelltextstelle an - vorausgesetzt er findet sie. Zum Beispiel kann damit eine Adresse eines Enforcerhits oder eines Absturzes schnell gefunden

werden, insbesondere Hunknummer und -offset, die ja unabhängig von der absoluten Adresse des Programms sind und durch Programme wie *SegTracker* ausgegeben werden, können gute Dienste in der Fehlersuche leisten.

Der letzte Menüpunkt **„Programm abbrechen"** ist etwas mit Vorsicht zu genießen und wird daher auch mit einer zusätzlichen Rückfrage quittiert. Mit diesem Menüpunkt läßt sich das Programm an jeder beliebigen Stelle abbrechen, solange es nicht gerade läuft. Hat man einen schweren Fehler gefunden, und steht nun auf einem Breakpoint, und ist sicher, daß bei der weiteren Programmausführung das Programm abstürzen wird, kann es sinnvoll sein, das Programm frühzeitig abzubrechen. Allerdings werden dadurch keine Dateien oder Fenster geschlossen, Speicher, oder irgendwelche anderen Ressourcen des Betriebssystems freigegeben.

Das Menü Fenster

Dieses Menü wurde schon in der Fensterbeschreibung erläutert. Je ein Menüpunkt öffnet eines der Fenster des Debuggers.

Das Menü Einstellungen

Mit dem Menüpunkt **„Einstellungen laden..."** kann eine neue Konfiguration der Oberfläche geladen werden.

Der Menüpunkt **„Einstellungen sichern"** speichert die aktuelle Konfiguration als Standardkonfiguration (unter dem Namen *„PROGDIR:MSDB.prefs"*).

Der Menüpunkt **„Einstellungen sichern als..."** speichert die aktuelle Konfiguration unter dem Namen, den Sie im Dateidialog eingeben.

Anhang

Der Enforcer

Der Enforcer ist ein Programm, mit dem illegale Speicherzugriffe auf Rechnern mit MMU verfolgt werden können. Viele Programmierfehler in C äußern sich auf diese Art und Weise (wegen der intensiven Benutzung der Pointer in C). Klassisch ist ein Enforcer-Hit (so wird ein erkannter illegaler Speicherzugriff genannt) in den unteren 20 oder 40 Bytes des Adreßraums, zumeist eine Folge eines Zugriffs auf den Speicher über einen nicht initialisierten Pointer, der den Wert NULL hat. Der Einsatz des Enforcers stört MaxonSDB nicht, um selbst Enforcerhits zu vermeiden, testet der Debugger jeden Speicherzugriff auf eine Variable des laufenden Programms auf den gültigen Adreßraum. Allerdings ist derzeit dieser Test noch stark eingeschränkt auf den wirklichen Speicher, der in Speicherlisten eingetragen ist, alle anderen Zugriffe werden unterdrückt. Um einen Enforcerhit im eigenen Programm zu finden, genügt es, den PC, an dem der Enforcerhit ausgelöst wurde, im Adreßfenster einzugeben, sollte diese Adresse im Programm liegen, wird der Quelltext korrekt angezeigt.

Der SegTracker

Der SegTracker arbeitet mit Enforcer zusammen, indem er die Adressen der Enforcerausgaben in die Hunknummern und -offsets des Programms umrechnet, das die Adresse enthält. Da im Adreßfenster auch Hunknummern und -offsets eingegeben werden können, und SegTracker auch die Umsetzung der Register und des Stacks beherrscht, können damit auch leicht Enforcer-Hits lokalisiert werden, die zwar durch ihr Programm verschuldet sind, aber erst durch das Betriebssystem ausgelöst werden.

inline-Funktionen

Um die beim Aufruf und bei Rückkehr einer Funktion notwendigen Maschinenbefehle einsparen zu können, wurden in C++ die inline-Funktionen eingeführt. Für diese Funktionen wird nicht nur wie bei anderen Funktionen einmal Code erzeugt, der dann bei jedem Funktionsaufruf angesprungen wird. Statt dessen merkt sich der Compiler die Zeilen der Funktion und ersetzt (fast) jeden Aufruf der Funktion durch den Funktionskörper. (Es gibt spezielle Fälle, wo diese Ersetzung nicht möglich ist, z.B. beim Aufruf der Funktion über einen Funktionspointer oder bei virtuellen Funktionen.) Analog der `register`- Anweisung in normalem C ist die Beachtung des Schlüsselwortes `inline` keine Pflicht für den Compiler, er kann das Wort schlicht ignorieren.

Durch die Verwendung von `inline` bei Funktionen wird ein Programm im allgemeinen nicht kürzer, sondern höchstens schneller. Deshalb lohnt es sich auch nur dann, Funktionen mit der Eigenschaft `inline` zu versehen, wenn diese sehr kurz sind, z.B. weil sie nur einen Wert aus einem Klassenobjekt zurückliefern. Bei einem global optimierenden Compiler allerdings kann die Verwendung des Schlüsselwortes auch ein kürzeres Programm zur Folge haben, da sich diesem neue Möglichkeiten zur Optimierung über Funktionsaufrufe hinweg bieten.

Für einen Debugger hat die Verwendung von `inline` mehrere, unangenehme Konsequenzen, die daraus resultieren, daß aus einem kurzen Stück Quelltext viele verstreute Stellen im Code werden, und somit keine eindeutige Zuordnung „Quelltext zu Code" mehr möglich ist.

Dadurch ist es nicht möglich, im Quelltext einen Haltepunkt in eine `inline`-Funktion zu setzen, und in Assembler müßten Sie einen Extra-Haltepunkt in jedes expandierte Stück setzen.

Eine `inline`-Funktion baut keinen Stack-Rahmen auf und verwendet die Register jedesmal anders, so daß ihre lokalen Variablen dem Debugger nicht zugänglich sind.

Der tiefere Sinn von `inline`-Funktionen, daß sie einem global optimierenden Compiler neue Optimierungsmöglichkeiten eröffnen, kann auch nicht genutzt werden, da ein solcher Compiler bei seiner Arbeit Quellzeilen und Variablen vertauscht, wegläßt und in ihrer Bedeutung verändert, und einen Quelltext-Debugger dabei vor kaum lösbare Probleme stellt. (Es ist sinnvoller, einen C++-Interpreter zu schreiben, als einen Debugger, mit dem man den Code im Quelltext nachvollziehen kann, den ein guter global optimierender C++-Compiler erzeugt).

MaxonSDB löst das Problem der `inline`-Funktionen folgendermaßen: Sie werden genau wie Makros behandelt. Dies bedeutet, daß der Aufruf einer `inline`-Funktion am Stück ausgeführt wird, auch wenn Sie **„Schritt/Überschreiten"** gewählt haben.

Technische Einzelheiten:

Der Debugger startet das Programm als eigenen Prozess und kommuniziert dann über Nachrichten mit ihm. Der Vorteil dieser Methode liegt darin, daß das Programm den Debugger nicht stört, und umgekehrt. Einzige Einschränkung: Dem Programm steht beim Debuggen ein Signal weniger zur Verfügung, als beim freien Lauf. Allerdings sollte kein Programm alle 16 freien Signale benötigen, oder ein Signal mit einer bestimmten Nummer belegen. Ein weiterer Vorteil ist die Möglichkeit, auch in ein laufendes Programm Breakpoints setzen, oder Variablen inspizieren, oder in jeder anderen Art mit dem Debugger weiter arbeiten zu können. Ein Nachteil soll allerdings auch nicht verschwiegen werden: DieKommunikation zwischen den Prozessen kostet ihre Zeit, und so kann die Abarbeitung eines Einzelschritts, der viele einzelne Maschinenbefehle bedeutet, eine kurze Weile dauern.

Forbid() und Disable()

Falls ein durch diese Funktionen geschützter Bereich in Einzelschritten abgearbeitet wird, wird dieser Schutz bei jedem Schritt durchbrochen, da Ihr Programm innerhalb von MaxonSDB `Wait()` aufruft (aufrufen muß!), um den Befehl zum nächsten Einzelschritt von der Benutzeroberfläche von MaxonSDB zu erhalten. Dieses Problem würde aber genauso auftreten, wenn MaxonSDB nur einen Prozeß für seine Tätigkeit verwenden würde, da der Debugger ja letztendlich auf Intuition warten muß, um das von Ihnen eingegebene Einzelschrittkommando zu erhalten.

Ein mit obigen Funktionen geklammerter Programmabschnitt, der mit **„Einzelschritt"** oder **„Überschreiten"** abgearbeitet wird, bleibt aber geschützt, falls Ihr Programm in diesem geschützten Abschnitt nicht auf einen Haltepunkt stößt. Auch innerhalb von Funktionen, die mit **„Überschreiten"** am Stück ausgeführt werden, bleibt der Schutz durch `Forbid()` oder `Disable()` erhalten.

Die Prozessoren 68020,68030 und 68040

MaxonSDB berücksichtigt automatisch den in Ihrem Amiga gerade benutzten Prozessor (so wie es jedes Programm tun sollte und leicht kann). Darüberhinaus kümmert er sich um den Cache dieser Prozessoren, wenn er z.B. Haltepunkte im Speicher installiert oder entfernt. Unter Kickstart 2.0 verwendet er die hierfür vorgesehenen Funktionen der Exec-Library, so daß es auch auf Amigas mit zusätzlichem Cache keine Probleme geben sollte (falls es solche Amigas überhaupt gibt).

Sicherheitsabfragen

Programm normal beendet

Das debuggte Programm wurde normal beendet. Der Dialog erlaubt außer dem Klick auf ***„OK"*** auch die Antwort ***„Debugger verlassen"*** woraufhin der Debugger beendet wird.

Es läuft noch ein Programm. Bitte beenden Sie dieses zuerst.

Bevor Sie ein neues Programm laden können, müssen Sie zuerst das laufende Programm beenden.

Abhilfe: Im Normalfall sollten Sie das Programm beenden, indem Sie das Programm mit **„Gehe"** (ohne gesetzte Breakpoints) schnell ablaufen lassen, bis es sich selbst beendet. Sie können das Programm auch mit dem Menüpunkt **„Programm abbrechen"** beenden

Programm wirklich abbrechen? Dadurch werden die Ressourcen...

Wenn Sie das Programm abbrechen werden die Ressourcen (Fenster, Speicher, Dateien,...) des Programmes nicht automatisch freigegeben oder geschlossen. Insbesondere bei Fenster kann es zu Systemabstürzen kommen, wenn ein freigegebenes aber immer noch dargestelltes Fenster aktiviert wird.

MaxonSDB kann noch nicht verlassen werden, da das...

Diese Meldung erscheint, wenn Sie den Debugger beenden wollen, bevor das debuggte Programm beendet wurde.

Abhilfe: Im Normalfall sollten Sie das Programm beenden, indem Sie das Programm mit **„Gehe"** (ohne gesetzte Braekpoints) schnell ablaufen lassen, bis es sich selbst beendet. Sie können das Programm auch mit dem Menüpunkt **„Programm abbrechen"** beenden

Fehlermeldungen

Es ist eine Ausnahme aufgetreten.
Das Programm muß vorzeitig beendet werden.

Der Debugger fängt jede Ausnahme (Exception) der CPU ab, wodurch das Programm beendet wird.

Abhilfe: Mit Einzelschritten den genauen Fehlerort suchen und das Programm korrigieren

Die Datei „xxx" konnte nicht geladen werden.

Diese Fehlermeldung tritt immer dann auf, wenn eine Quelltextdatei nicht geladen werden konnte. Hierfür gibt es mehrer Ursachen:

1. Die Quelltext-Datei wurde nach dem Compilieren gelöscht oder die Diskette befindet sich nicht im Laufwerk.

2. Der Debugger wurde aus einem anderen Verzeichnis als der Compiler gestartet. Die MDBI-Dateien (die unter anderem alle Quelltextnamen zu einem Modul enthalten), zeigen häufig mit relativen Pfadnamen auf die Quelltextdateien. Wurde der Debugger aus einem anderen Verzeichnis als der Compiler gestartet, ist es wahrscheinlich, daß der Debugger die Quelltextdateien nicht finden kann.

Insbesondere beim Debugger-Start aus der Shell kann es leicht passieren, daß Compiler und Debugger aus verschiedenen Verzeichnissen gestartet werden.

Abhilfe: Überprüfen Sie ob die Quelltext-Datei vorhanden ist und starten Sie den Debugger aus dem gleichen Verzeichnis aus dem Sie auch den Compiler starten.

Das Programm „xxx" konnte nicht geladen werden.

Das zu debuggende Programm konnte nicht geladen werden. Entweder steht keine ausführbare Datei mit dem angegebenen Namen zur Verfügung oder es ist keine passende MDBG-Datei vorhanden.

Abhilfe: Stellen Sie sicher, daß eine Exe-Datei und die Debug-Dateien erzeugt werden und kompilieren das Programm neu.

Das Programm „xxx" konnte nicht gestartet werden.

Das korrekt geladene Programm konnte nicht gestartet werden. Dieser Fehler kann nur durch Speicherplatzmangel auftreten.

AMIGA

MaxonC++

HotHelp 3

MAXON
computer

1. Über diese Anleitung

HotHelp liegt mittlerweile in der Version 3 vor. Falls Sie schon mit einer früheren Version des Programms vertraut sind, wird Sie die folgende Einführung und das Kapitel über die Bedienung sicherlich nicht besonders interessieren, da diese sich an den HotHelp-Einsteiger wenden. Um so interessanter dürfte für Sie jedoch das Kapitel sein, in dem die Neuerungen und Änderungen gegenüber Version 2 zusammengefaßt sind - dort sollten Sie unbedingt einen (oder auch zwei oder mehr) Blicke hineinwerfen!

Sie werden im übrigen feststellen, daß die Anleitung nicht mehr so detailliert auf HotHelp eingeht wie bei Version 2 - das ist auch nicht mehr notwendig, da HotHelp sich inzwischen komplett selbst erklärt! Das Handbuch beschreibt daher nur noch die Fähigkeiten und größeren Zusammenhänge innerhalb des HotHelp-Systems - für Detailfragen, die sich erst während der Benutzung eines Programms stellen (was genau macht der Schieberegler hier und wofür ist dieses Blättersymbol dort nützlich?) kann dann jederzeit ein Hilfstext abgerufen werden.

Sollten sich nach Drucklegung dieser Anleitung noch kurzfristig Änderungen an den Programmen ergeben haben, finden Sie auf der ersten Installations-Diskette noch eine Datei mit dem Namen "Liesmich", die Sie sich ebenfalls ansehen sollten.

2. Jederzeit hilfsbereit

Vielleicht gehören Sie ja zu den Leuten, die alles, was sie zum Arbeiten mit dem Amiga brauchen, in einem geradezu gigantischen Gedächtnis gespeichert haben. In diesem Fall haben Sie sich HotHelp leider umsonst gekauft - sorry! Verfügen Sie jedoch über kein fotografisches Gedächtnis, dann brauchen Sie HotHelp!

HotHelp ist eine Art elektronische Bibliothek, die Ihnen die gesuchten Informationen genau da zur Verfügung stellt, wo Sie sie auch benötigen - nämlich am Computer! Das Programm kann jederzeit über einen einfachen Tastendruck aktiviert werden und öffnet dann ein Fenster, in dem Sie sich die einzelnen Hilfstexte anzeigen lassen können.

Wie in einer Bibliothek gibt es verschiedene Bücher, die hier als Projekte bezeichnet werden. Ähnlich einem normalen Lexikon enthält jedes Projekt eine Reihe von Hilfstexten zu ganz unterschiedlichen Stichworten (diese werden hier als Schlüssel bezeichnet). Jeder Text kann noch eine Reihe von Querverweisen auf verwandte Themen enthalten; wird ein solcher Verweis mit der Maus oder der Tastatur ausgewählt, stellt HotHelp den dazu passenden Hilfstext dar (Hypertext-Prinzip). Entgegen einem herkömmlichen Lexikon übernimmt dabei jedoch der Amiga die umständliche Suche und Blätterei, so daß Sie sich ganz auf die Texte konzentrieren können!

HotHelp bietet dabei für jeden Amiga-Anwender etwas:

- Der Einsteiger findet im Glossar Erklärungen zu mehr als 500 verschiedenen Begriffen des typischen Amiga-Fachchinesisch (bzw. -englisch). Wenn Sie sich schon immer gefragt haben, welche Vorteile Ihnen Multitasking eigentlich bietet, was ein Guru mit dem Amiga zu tun hat, was die Aufgabe eines Betriebssystems ist, worum es sich bei einem Gadget oder einem Drawer handelt oder was bei einem Absturz genau passiert, finden Sie hier die Antworten auf Ihre Fragen.
- Für den fortgeschrittenen Amiga-Benutzer, der seinen Computer nicht nur über die Workbench steuern möchte, ist sicherlich die ausführliche Beschreibung aller Shell-Befehle und die Erklärung der Programmiersprache ARexx interessant (letztere ist schon seit einiger Zeit Bestandteil des Amiga-Betriebssystems und hat das veraltete AmigaBASIC abgelöst).
- Programmierer finden ausführliche Informationen über die Programmiersprache C (ANSI-Standard), den IFF-Standard sowie eine umfassende Dokumentation aller Betriebssystem-Funktionen, -Strukturen und -Konstanten bis einschließlich OS 3.0. Zusammen mit der Möglichkeit, Hilfe zum aktuellen Wort im gerade editierten Text abzurufen, bietet HotHelp damit eine enorme Arbeitserleichterung, da das lästige Nachschlagen in den Amiga-Handbüchern völlig entfällt!

3. Eigenschaften von HotHelp

Neben HotHelp, das jetzt in der neuen Version 3 vorliegt, gibt es noch einige andere ähnliche Programme auf dem Markt - unter anderem auch das von Commodore favorisierte AmigaGuide-System, das neuerdings auch Bestandteil des Betriebssystems ist. Hier soll nicht auf die verschiedenen Vor- oder Nachteile der anderen Produkte eingegangen werden (schließlich ist vergleichende Werbung ja auch verboten) - sie finden hier einfach eine Übersicht über alle Vorzüge von HotHelp. Vergleichen Sie diese ruhig einmal mit den Leistungsmerkmalen der Konkurrenzprodukte; Sie werden feststellen, daß HotHelp der Porsche unter den Hilfssystemen für den Amiga ist!

- HotHelp wird mit umfangreichen Hilfstexten zu einer ganzen Reihe interessanter Themen ausgeliefert (Glossar, Shell, ARexx, ANSI-C, IFF, Libs & Devs 3.0). Da die neue Version auch Dateien im AmigaGuide-Format akzeptiert, können auch die Hilfsdateien anderer Programme, die in diesem Format vorliegen, über HotHelp angezeigt werden.

- HotHelp ist auf allen Betriebssystemen ab Version 1.2 lauffähig. Alle Programme wurden anhand der von Commodore vorgeschriebenen Entwicklungsrichtlinien erstellt, mit Hilfe der Commodore-Testprogramme überprüft und haben sich als ausgesprochen stabil erwiesen.

- HotHelp-Fenster können aus jeder systemkonform programmierten Anwendung heraus aufgerufen werden und stehen somit während der Arbeit am Amiga jederzeit zur Verfügung, sobald sie benötigt werden. HotHelp unterstützt jede Bildschirmauflösung und jeden Zeichensatz fester Breite; der Fließtext paßt sich an jede Fensterbreite optimal an. Auf Wunsch kann das Programm Zeichensatz und Farben automatisch an den aktuellen Bildschirm anpassen.

- In HotHelp-Hilfstexten können Texte und Grafiken miteinander vermischt werden, wodurch die Texte wesentlich attraktiver gestaltet werden können.

- Hilfstexte können ausgedruckt, in den aktuellen Text Ihres Editors, den Zwischenspeicher (Clipboard) oder eine Datei geschrieben sowie direkt als Eingabe in ein laufendes Programm verwendet werden.

- Auf Tastendruck hin kann ein Hilfstext zum Wort unter dem Cursor in den Editoren Edward (MAXON), CygnusEd (CygnusSoft) oder TurboText (Oxxi) abgerufen werden. Diese Möglichkeit ist speziell für Programmierer sehr interessant. Außerdem kann Hilfe zum aktuellen Inhalt des Zwischenspeichers angefordert werden.

- HotHelp bietet die einzigartigen Onlineprojekte, bei denen Hilfstexte ohne den Umweg über die Übersetzung per Knopfdruck aus einem Editor heraus definiert und jederzeit wieder abgerufen werden können.

- Durch Komprimierung der Hilfstexte werden Diskettenplatz und Ladezeiten gespart.

- Die Erstellung eigener Projekte mit Hilfstexten ist nun durch zwei neue Zusatzprogramme sehr einfach geworden. Sie können dort mit Hilfe einer editor-ähnlichen grafischen Oberfläche über

Maus oder Tastatur alle erforderlichen Kennungen in den Text einfügen, die HotHelp benötigt, um z.B. Schlüsselbegriffe vom übrigen Text unterscheiden zu können. Sie finden dazu eine ausführliche Einführung in die Erstellung von Projekten, die Ihnen Schritt für Schritt erklärt, wie Sie Hilfsdateien mit eigenen Texten erzeugen können.

☞ Die Geschwindigkeit der Übersetzung selbst wurde gegenüber der letzten Version mehr als verdoppelt. Auch die Überprüfung eigener Projekte läuft jetzt mehr als doppelt so schnell ab wie bisher; das Testmodul wurde erweitert und meldet nun (neben dem Hinweis auf fehlende Schlüssel) noch eine Reihe anderer möglicher Fehlerquellen.

☞ Für die Darstellung von Quelltexten, Tabellen und ähnlichem in einem HotHelp-Projekt kann die Fließtext-Darstellung abgeschaltet werden.

☞ Das neue Ansichten-Konzept ermöglicht es, mehrere Texte gleichzeitig in einem HotHelp-Fenster zu halten und über Maus oder Tastatur zwischen den Texten hin- und herzuwechseln. Die verschiedenen Textansichten sind vollkommen unabhängig voneinander.

☞ Die ARexx-Schnittstelle erlaubt es, HotHelp von jedem ARexx-fähigen Programm aus anzusteuern.

☞ Über die neue Suchfunktion können Sie jetzt beliebige Begriffe innerhalb eines Textes suchen. Darüber hinaus haben Sie nun auch die Möglichkeit, alle Texte eines Projektes (oder auch nur einen bestimmten Teil der Texte) nach einem beliebigen Begriff zu durchsuchen, der nicht als Schlüssel innerhalb des Projektes definiert sein muß.

☞ Es können bis zu fünf Lesezeichen gesetzt werden, zu denen Sie später wieder auf Tastendruck zurückkehren können.

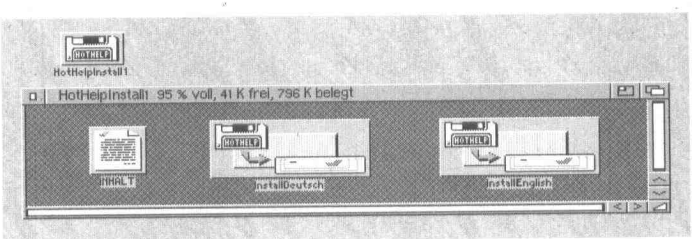
☞ Die Auswahl mehrerer Projekte und Schlüssel gleichzeitig wird durch die Verwendung von Namensmustern ermöglicht, wie sie auch in der Shell verwendet werden.

4. Anforderungen

HotHelp benötigt einen Amiga mit Kickstart 1.2 oder höher, zwei Diskettenlaufwerken und 512 KByte Speicher. Sollen alle Projekte installiert werden, sollte 1 MByte Speicher und eine Festplatte vorhanden sein - ansonsten wäre dauernder Diskettenwechsel die Folge!

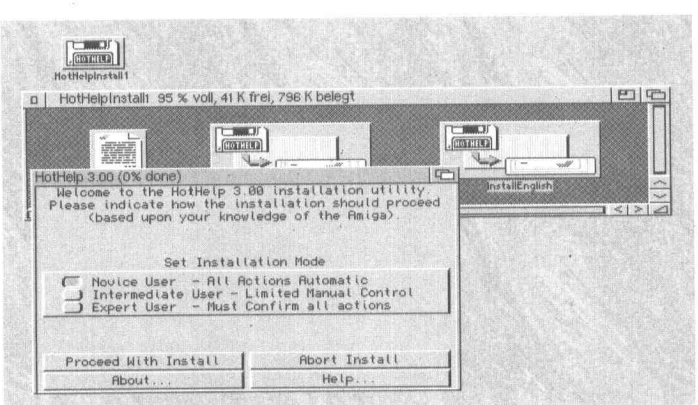
5. Installation

Damit HotHelp Ihnen jederzeit auf Tastendruck zur Verfügung steht, muß es auf Ihrer normalen Startdiskette bzw. Festplatte installiert werden. Zu diesem Zweck wird das Installationsprogramm 'Installer' von Commodore verwendet. Es ist sehr einfach zu bedienen, da es sich der Amiga-Erfahrung des Anwenders anpassen kann.



5.1. Installation auf Festplatte

Wenn Sie eine Festplatte besitzen, sind keine weiteren Vorkehrungen erforderlich. Sie sollten sich lediglich vergewissern, daß auf der Platte noch mindestens 4 MByte freier Speicherplatz vorhanden ist - soviel benötigt das komplette HotHelp-System in etwa.



5.2. Installation auf Disketten

Anwender ohne Festplatte können die volle Leistungsfähigkeit von HotHelp kaum ausnutzen - schließlich benötigen die gesamten Hilfsdateien schon fast zwei MByte an Speicher (das sind über zwei Millionen Zeichen). Allein diese Informationen belegen also schon drei Disketten! Möchten Sie dann wirklich Hilfstexte aus allen Projekten abfragen, werden Sie schnell zum Diskjockey.

Als Einsteiger (mal vorausgesetzt, daß alle Profis Festplatten besitzen...) haben Sie aber z.B. sicher kein Interesse an den Projekten, die sich lediglich mit Fragen der Programmierung beschäftigen, so daß Sie auf die Installation der entsprechenden Projekte verzichten können. Dadurch reduziert sich der Platzbedarf ganz erheblich! Sie haben während der Installation die Gelegenheit, unerwünschte Projekte auszuschließen. Nutzen Sie diese Möglichkeit!

Je nachdem, wieviel Daten Sie installieren wollen, werden bis zu vier formatierte Leerdisketten benötigt (Hinweise zum Formatieren von Disketten entnehmen Sie bitte Ihrem Amiga-Benutzerhandbuch). Stellen Sie sicher, daß alle Disketten unterschiedliche Namen tragen (z.B. "HotHelp 1", "HotHelp 2", ...) und daß keine Disk den Namen "HotHelp" trägt. Sie sollten hier keinen Fehler machen, da ansonsten später unangenehme und schwer zu behebende Probleme auftreten können.

Weiterhin müssen Sie noch sicherstellen, daß auf Ihrer Start-Diskette (das ist die Disk, die Sie üblicherweise nach dem Einschalten des Amiga einlegen, um das System zu starten) noch mindestens 130 KByte freier Speicherplatz verfügbar sind. Sie können dies erreichen, indem Sie unbenutzte Programme löschen oder selten benutzte auf andere Disketten auslagern.

Haben Sie schließlich alle Vorbereitungen abgeschlossen, starten Sie Ihren Amiga bitte mit der bereinigten Startdiskette neu.

5.3. Der Installationsvorgang

Um die Installation zu starten, legen Sie die HotHelp-Diskette 1 bitte in ein beliebiges Laufwerk und öffnen die Disk durch einen Doppelklick auf das erscheinende Piktogramm. Die Disk enthält drei weitere Piktogramme: 'INHALT', 'InstallDeutsch' und 'InstallEnglish'. Die Notwendigkeit für zwei Installationsprogramme ergibt sich aus der Tatsache, daß der Amiga erst seit kurzem die Verwendung unterschiedlicher Sprachen gestattet. Die beiden Programme unterscheiden sich daher nur in einer Hinsicht: in der englischen Version sind die Beschriftungen von Symbolen und bestimmte Standardtexte in englischer Sprache gehalten, während in der deutschen Version alle Texte in Deutsch erscheinen. Installiert wird in beiden Fällen dasselbe HotHelp!

Als Faustregel gilt: wenn Sie auf Ihrer Workbench (z.B. in der Titelzeile) deutsche Texte sehen, können Sie auch die deutsche Installation verwenden - ansonsten müssen Sie leider auf die englische zurückgreifen. Falls Sie mit dem Englischen nicht vertraut sind, finden Sie am Ende dieses Kapitels einen Abschnitt, in dem die wichtigsten Begriffe, die während der Installation auftreten können, erklärt werden.

Starten Sie nun die Installation durch einen Doppelklick auf das entsprechende Piktogramm. Nach einer einleitenden Nachricht erreichen Sie das Startfenster, in dem Sie den Modus der Installation festlegen müssen. Sie haben dabei die Möglichkeit, zwischen 'Einsteiger', 'Geübter Benutzer' und 'Experte' zu unterscheiden.

Im Einsteiger-Modus wird davon ausgegangen, daß Sie über eine Festplatte verfügen, auf die HotHelp dann installiert wird. Sie müssen dazu anschließend noch den Namen dieser Platte eingeben. Es handelt sich dabei um den Namen, der auch auf der Workbench unter dem Piktogramm Ihrer Festplatte sichtbar ist, gefolgt von einem Doppelpunkt. Alle weiteren Aktionen werden dann automatisch durchgeführt - Sie werden lediglich noch aufgefordert, entsprechende Disketten einzulegen.

Im Modus für geübte Benutzer muß eine Reihe von Abfragen beantwortet werden - welche Funktionsbibliotheken installiert werden sollen, in welchem Verzeichnis die Daten für HotHelp abgelegt werden müssen und ähnliches. Normalerweise können Sie die vom Programm gemachten Vorgaben einfach akzeptieren. Sie sollten diesen Modus verwenden, wenn Sie mehr Kontrolle über den Verlauf der Installation wünschen - dieser Modus ermöglicht es z.B. auch, HotHelp auf Disketten zu installieren.

Im Experten-Modus schließlich muß jede einzelne Aktion bestätigt werden. Sie haben dadurch die volle Kontrolle über den Installationsvorgang, werden jedoch auch mit einer Reihe eigentlich unwichtiger Details belästigt.

Falls Sie die Modi für geübte Benutzer oder Experten verwenden, können Sie in den meisten Fenstern über das Hilfe-Symbol einen ausführlichen Hilfstext über den aktuellen Status und den Sinn des entsprechenden Fensters abfragen. Scheuen Sie sich nicht (auch wenn Sie als Experte installieren), dieses Symbol zu betätigen, wenn Ihnen die Bedeutung der aktuellen Abfrage für die Installation nicht ganz klar sein sollte.

Unabhängig vom eingestellten Modus geht das Installationsprogramm möglichst genau auf Ihre aktuelle Konfiguration ein. Insbesondere wird auch erkannt, wenn Sie HotHelp schon einmal installiert haben. Das Skript sorgt dann dafür, daß die alte HotHelp-Version problemlos durch die neue ersetzt wird.

Wenn Sie die Installation erfolgreich beendet haben, müssen Sie Ihren Rechner neu starten (Ctrl-Amiga-Amiga), um HotHelp zu aktivieren. Das Programm steht Ihnen von nun an nach jedem Neustart automatisch zur Verfügung. Sie erkennen dies daran, daß sich während des Starts kurzzeitig ein Fenster mit dem Text 'HotKeyHelp wird installiert' öffnet.

5.4. Installation von Hand

Manche Experten trauen ja grundsätzlich keinem Installations-Skript und wollen immer alles von Hand (bzw. von Shell) kopieren. Davon sei Ihnen jedoch an dieser Stelle entschieden abgeraten! Das Installations-Skript ist nicht umsonst recht lang geworden - viele Elemente von HotHelp sind auf den Installationsdisks noch komprimiert, die Projekte müssen noch nachbearbeitet werden, es muß eine Voreinstellungsdatei erzeugt, eine Änderung muß in die Datei 'User-Startup' eingetragen werden und noch einiges mehr.

Hier nur noch ein paar Hinweise auf die Aktionen während der Installation:

Ins Verzeichnis LIBS: werden zwei oder drei Libraries kopiert. Das Verzeichnis L: nimmt zwei Binärdateien, die sogenannten HotHelp-Handler, auf. Das ausführbare Programm HotKeyHelp wird ins C:-Verzeichnis kopiert. Wenn Ihr Betriebssystem schon verschiedene Sprachen unterstützt, werden zwei Katalogdateien in das Verzeichnis LOCALE:Catalogs/Deutsch übertragen. Die ausgewählten HotHelpTools, die Beispiel-Projekte und -Programme werden in den dafür ausgewählten Verzeichnissen abgelegt; alle übrigen Dateien werden in dem für die HotHelp-Daten vorgesehenen Verzeichnis untergebracht. Schließlich werden noch einige Befehle in die Datei 'User-Startup' eingebaut. Dabei wird das logische Gerät HOTHELP: dem Verzeichnis mit den HotHelp-Daten zugeordnet.

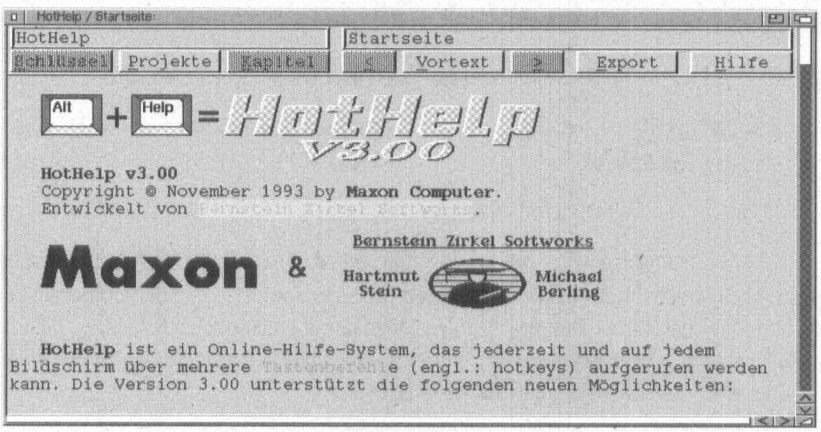
5.5. Installation mit englischer Benutzerführung

Falls Sie noch mit Betriebssystem-Versionen vor 2.1 arbeiten, unterstützt Ihr Amiga lediglich die englische Sprache. In diesem Fall müssen Sie leider das Installationsskript 'InstallEnglish' benutzen, in dem alle Symbole in Englisch beschriftet sind. Hier soll dazu kurz auf die wichtigsten Begriffe eingegangen werden.

- Proceed, Proceed With Install: Fortsetzen der Installation. Falls im aktuellen Fenster Einstellungen vorgenommen werden konnten, werden diese übernommen und der nächste Schritt der Installation eingeleitet.
- Abort, Abort Install: Abbruch der Installation. Hiermit wird der gesamte Installationsvorgang vorzeitig beendet.
- Help: Zeigt einen (deutschen) Text mit Informationen über das aktuelle Fenster an.
- Novice User, Intermediate User, Expert User: Dies sind die drei Installationsmodi Einsteiger, geübter Benutzer und Experte.
- Yes, No: Ja bzw. Nein bei einer einfachen Abfrage.
- Selected Drawer, Parent Drawer, Make new Drawer, Show Drives: Diese Begriffe tauchen bei der Auswahl einer Schublade (Drawer) auf. 'Selected Drawer' ist der Name der momentan eingestellten Schublade; 'Parent Drawer' zeigt den Inhalt der übergeordneten Schublade (Mutterverzeichnis) an, über 'Make new Drawer' können Sie eine neue Schublade erzeugen und 'Show Drives' zeigt eine Übersicht aller angeschlossenen Geräte an.

6. Die Bedienung von HotHelp

Nachdem die Installation abgeschlossen wurde und Sie einen Neustart ausgeführt haben, brennen Sie sicher schon darauf, das erste HotHelp-Fenster zu sehen. Drücken Sie dazu eine der beiden Alt-Tasten (am unteren Rand der Tastatur neben den Tasten mit den Amiga-Symbolen). Halten Sie die Taste gedrückt und drücken Sie noch zusätzlich die Help-Taste (oberhalb der vier mit Pfeilen beschrifteten Cursortasten). Schon sehen Sie Ihr erstes HotHelp-Fenster vor sich auf dem Bildschirm auftauchen.



Eine solche Tastenkombination wird als Hotkey (heiße Tasse ...äh... Taste) bezeichnet, da sie augenblicklich eine Reaktion hervorruft. Sie können diesen Hotkey jederzeit verwenden, um ein Hilfsfenster zu öffnen. Das Fenster erscheint immer auf der Arbeitsfläche des Programms, mit dem Sie gerade arbeiten.

ACHTUNG:

Einige Programme verwenden ihre Arbeitsfläche, ohne damit zu rechnen, daß diese auch von anderen Programmen benutzt wird. Aus diesem Grund kann es vorkommen, daß der Bildschirmaufbau des Programms mehr oder weniger stark durcheinander kommt (Datenverluste durch Abstürze sollten dadurch allerdings nicht auftreten).

Bei neueren Programmen können Sie das Problem beheben, indem Sie das Programm anweisen, seinen Bildschirm der Öffentlichkeit zur Verfügung zu stellen. Dafür müßte eine Einstellmöglichkeit wie z.B. 'Bildschirm öffentlich machen' oder 'Make screen public' vorhanden sein. Sollte das von Ihnen verwendete Programm keine solche Möglichkeit bieten, sollten Sie HotHelp nur noch vorsichtig auf dem Bildschirm dieses Programms einsetzen!

ACHTUNG:

Schließen Sie niemals einen Bildschirm, auf dem sich noch ein geöffnetes HotHelp-Fenster befindet, da Sie HotHelp damit den Boden unter den Füßen wegziehen! Dies hat meist einen bodenlosen Absturz zur Folge...

6.1. Die Titelzeile des HotHelp-Fensters

Das neu geöffnete Fenster zeigt Ihnen einen einleitenden Text über HotHelp selbst an. In der Titelzeile des Fensters sehen Sie den Titel dieses Textes - es handelt sich um die Startseite aus dem HotHelp-Projekt. Um wieder den Bibliotheksvergleich zu bemühen: Sie sehen jetzt die Einleitung des Buches, das sich speziell mit den Fähigkeiten von HotHelp befaßt. Wir können also festhalten: in der Titelzeile wird immer angezeigt, welchen Text Sie gerade lesen.

6.2. Rollsymbole

Im rechten und unteren Rand des Fensters sehen Sie zwei Rollsymbole vor sich, die dazu dienen, den sichtbaren Textausschnitt einzustellen. Das Symbol im unteren Fensterrahmen wird dabei nur benötigt, wenn Sie das Fenster so weit verkleinern, daß ein Text nicht mehr in eine sichtbare Zeile paßt. Da HotHelp bei der Ausgabe üblicherweise mit Fließtext arbeitet (dieser wird bei jeder Änderung der Fenstergröße so gut wie möglich an die neue Größe angepaßt), passiert dies nur relativ selten. Anders verhält es sich mit Grafiken - da diese natürlich nicht umgebrochen werden können, kommt hier das Rollsymbol im unteren Fensterrand häufiger zum Einsatz. Sie können das überprüfen, indem Sie jetzt das Fenster mit der HotHelp-Startseite auf halbe Breite bringen.

Wesentlich wichtiger ist das Rollsymbol im rechten Rand - es wird immer benötigt, wenn der Text mehr Zeilen beansprucht als im Fenster Platz sind (was bei den meisten Texten der Fall ist, wie auch bei der HotHelp-Startseite). Sie können sich dann mit Hilfe dieses Symbols den ganzen Text Zeile für Zeile anzeigen lassen.

Sie können den Text auch direkt über die Maus verschieben. Klicken Sie dazu einfach das Fenster ober- oder unterhalb des Textbereiches an: der sichtbare Textausschnitt verschiebt sich um eine Zeile. Halten Sie die Maustaste gedrückt, so wird der Rollvorgang nach kurzer Zeit automatisch wiederholt. Er bricht ab, sobald Sie die Taste wieder loslassen.

Eine weitere Möglichkeit besteht darin, eine beliebige Position im Textbereich anzuklicken und die Maustaste gedrückt zu halten. Ziehen Sie die Maus nun aus dem Textbereich heraus, wird der Text in die entsprechende Richtung verschoben.

6.3. Querverweise

Querverweise auf andere Texte bieten Ihnen eine einfache Möglichkeit, weiterführende Informationen abzurufen. Sie können an jeder Position im Text erscheinen und werden durch eine andere Farbe gekennzeichnet (normalerweise Blau ab Betriebssystemversion 2.0, sonst Orange). Einer der Querverweise wird immer invertiert dargestellt (also nicht Blau auf Grau, sondern umgekehrt) - dies ist der aktuelle Querverweis. Sie können diesen mit Hilfe der Maus oder der Tastatur setzen und den zugehörigen Text abfragen. Ein Querverweis kann sich auf einen Text im aktuellen oder einem anderen Projekt beziehen.

Bei Verwendung der Maus genügt es, einen Querverweis anzuklicken, um ihn zum aktuellen Querverweis zu machen. Wenn Sie irgendwo ins Fenster klicken, dann die linke Maustaste gedrückt halten und die Maus bewegen, können Sie jeden der sichtbaren Querverweise in beliebiger Reihenfolge anwählen. Verlassen Sie dabei mit dem Mauszeiger den Textbereich, so wird der Text in die entsprechende Richtung verschoben. Sie können also sehr einfach jeden Querverweis im Text ansteuern. Durch einen Doppelklick auf einen Querverweis wird der dazu passende Text von HotHelp nachgeladen und dargestellt.

Wollen Sie die Tastatur verwenden, wählen Sie bitte zuerst den gewünschten Querverweis aus. Sie können dazu die Tabulatortaste oder die vier Cursortasten benutzen. Gegebenenfalls verschiebt

HotHelp den sichtbaren Textausschnitt, um den aktuellen Querverweis anzuzeigen. Haben Sie schließlich den gewünschten Querverweis angewählt, können Sie sich durch Betätigen der Eingabetaste (Return oder Enter) den zugehörigen Text anzeigen lassen.

Es gibt noch eine weitere Methode, einen Querverweis über die Tastatur auszuwählen. Halten Sie dazu eine der beiden Alt-Tasten gedrückt und geben dann die Anfangsbuchstaben des Querverweises ein, so wird dieser von HotHelp zum aktuellen Querverweis gemacht, dessen Hilfstext Sie dann wie üblich mit der Eingabetaste abfragen können. Dies ist recht praktisch, wenn ein langer Text viele Querverweise enthält und Sie einen dieser Querverweise auswählen möchten, aber nicht wissen, wo er im Text eigentlich auftritt - überlassen Sie einfach HotHelp die Suche!

6.4. Das Vortext-Symbol

Im Gegensatz zu einem normalen Buch merkt sich HotHelp die Seiten, die Sie bisher gelesen haben. Über das Vortext-Symbol können Sie nun ohne weiteres wieder zu den entsprechenden Texten zurückkehren. Normalerweise merkt sich das Programm bis zu 20 Schritte; sie können diese Anzahl jedoch bis auf 500 erhöhen...

6.5. Blättern im Projekt

Wie in einem richtigen Buch können Sie auch in einem HotHelp-Projekt nach Herzenslust blättern. So, wie in einem Buch Seite auf Seite folgt, folgt in HotHelps Projekten Hilfstext auf Hilfstext. Sie können die beiden mit Pfeilen beschrifteten Symbole links und rechts vom Vortext-Symbol dazu benutzen, um die Texte des Projektes der Reihe nach anzusehen. Im Gegensatz zu einem Lexikon, wo alle Texte lediglich alphabetisch sortiert sind, finden Sie hier jedoch beim Blättern einen thematischen Zusammenhang vor. Sie können sich damit durch das Blättern in einem Projekt leicht einen Überblick über ein oder mehrere Themen verschaffen. Das Vortext-Symbol bringt Sie dabei immer wieder zu den zuletzt gelesenen Texten zurück.

6.6. Schlüssel, Projekte und Muster

Am oberen Rand des Fensters befinden sich zwei Eingabefelder. Das linke der beiden ist für den Projektnamen, das rechte für den Schlüsselbegriff zuständig. Wird ein neuer Hilfstext dargestellt, trägt HotHelp die entsprechenden beiden Begriffe in diese Eingabefelder ein.

Interessant werden die beiden Felder jedoch erst, wenn Sie selbst einen Text dort eingeben. Suchen Sie z.B. nach einem bestimmten Begriff, können Sie diesen in das rechte Eingabefeld eintragen. Bestätigen Sie dann die Eingabetaste, sucht HotHelp den zu dem Schlüssel passenden Text und stellt ihn dar. Sie haben so durch die direkte Eingabe die Möglichkeit, jeden gesuchten Begriff direkt anzugeben, ohne auf einen entsprechenden Querverweis angewiesen zu sein. Falls der Begriff in einem anderen als dem eingestellten Projekt vorhanden ist, sollten Sie den Namen des entsprechenden Projektes in das linke der beiden Felder eintragen.

Möglicherweise haben Sie manchmal das Problem, daß Ihnen der genaue Wortlaut des Suchbegriffs nicht bekannt ist. Nehmen wir einmal an, Sie wollen einen Hilfstext über das Stichwort 'Online-Pro-

jekte' abrufen. Leider ist Ihnen aber die genaue Schreibweise des Begriffs unbekannt - mögliche Kandidaten wären z.B. 'Online-Projekte', 'Onlineprojekt' oder auch 'Online Projekt'. Natürlich könnten Sie jetzt alle diese Begriffe ausprobieren (obwohl erfahrungsgemäß der einzig richtige Begriff der einzige ist, der Ihnen nicht einfallen wird). Das ist jedoch eine stupide, langweilige und frustrierende Aufgabe - also wie geschaffen dafür, von einem Computer übernommen zu werden!

HotHelp unterstützt zu diesem Zweck auch Namensmuster in den Eingabefeldern. Shell-Anwendern ist dieser Begriff sicherlich vertraut: über ein Muster kann bei den Shell-Befehlen eine ganze Reihe von Dateien gleichzeitig angesprochen werden. Der Trick liegt darin, daß in den Namen verschiedene Sonderzeichen eingesetzt werden, die ganze Gruppen von Zeichen oder Zeichenfolgen abdecken. Eines dieser Sonderzeichen ist der Stern (*) - er steht für einen beliebigen Text.

Geben Sie im obigen Beispiel daher einfach '*Online*Projekt*' als Schlüsselbegriff ein. HotHelp sucht dann alle Texte heraus, deren Schlüssel die Worte 'Online' und 'Projekt' (in dieser Reihenfolge) enthalten; zwischen den Worten dürfen sich dabei beliebige Zeichenfolgen befinden. Da diese Zeichenfolgen auch leer sein dürfen, finden Sie damit garantiert alle der oben angegebenen Schlüsselvariationen (der korrekte Begriff lautet in diesem Falle übrigens 'Onlineprojekt').

Dasselbe gilt auch für das Eingabefeld für den Projektnamen. Sind Sie sich nicht sicher, in welchem Projekt die gesuchte Information sich befindet, geben Sie hier einfach einen einzelnen Stern ein. HotHelp untersucht dann alle Projekte (denn jedes Projekt hat einen Namen aus einer beliebigen Zeichenfolge und stimmt somit mit dem angegebenen Muster überein) und zeigt alle gefundenen Schlüssel.

Natürlich kann es auch passieren, daß mehr als ein Schlüssel zu dem eingegebenen Muster paßt. In diesem Fall stellt HotHelp zuerst eine nach Projekten gegliederte Übersicht über alle Schlüssel auf, aus der Sie dann den entsprechenden Querverweis wie gewöhnlich auswählen können. Einen besonderen Komfort bietet Ihnen hier dann noch das Schlüssel-Menü, das Sie mit der rechten Maustaste erreichen. Über die beiden Einträge dieses Menüs können Sie der Reihe nach alle Texte durchblättern, die zu dem angegebenen Muster passen. Über das dann aktive Schlüssel-Aktionssymbol am linken Rand der Symbolleiste können Sie die Schlüsselübersicht jederzeit wieder abrufen.

Unter Betriebssystem-Versionen vor 2.0 steht Ihnen als Musterzeichen nur der einzelne Stern zur Verfügung. In höheren Versionen können Sie alle Shell-Namensmuster verwenden - schauen Sie dazu ggf. in Ihrem Amiga-Benutzerhandbuch nach, in dem Sie eine Beschreibung der Musterzeichen finden. Oder noch einfacher: sparen Sie sich diese mühsame Suche und rufen Sie lieber den Hilfstext im Projekt 'Glossar' unter dem Schlüsselbegriff 'Namensmuster' auf! Sie sind jetzt stolzer Besitzer von HotHelp - was sollen Sie noch mit Handbüchern...

6.7. Abrufen der Projektübersicht

Über das Projekte-Aktionssymbol können Sie eine Übersicht über alle geladenen Projekte abrufen. Sie sehen dann im Fenster eine Liste aller Projektdateien, aus der Sie eine wie üblich als Querverweis auswählen können. HotHelp stellt dann den Einleitungstext des entsprechenden Projektes dar.

6.8. Die Kapitelübersicht

HotHelp-Projekte sind üblicherweise nach Themengebieten gegliedert und in entsprechende Kapitel eingeteilt (wie es sich für ein gutes Buch gehört). Über das Kapitel-Aktionssymbol können Sie nun die zum aktuellen Text passende Kapitelübersicht anfordern.

Dieses Symbol ist nicht anwählbar, wenn Sie sich bereits auf der obersten Kapitelebene (der Projekt-Einleitung oder -Startseite) befinden.

6.9. Der Export von Hilfstexten

HotHelp wacht nicht eifersüchtig wie eine Glucke über ihre Eier (bzw. Hilfstexte), sondern ist durchaus bereit, diese mit anderen Programmen zu teilen. Sie haben sogar eine ganze Reihe von Möglichkeiten, um Texte an andere Programme zu übertragen. Als erstes müssen Sie dazu jedoch über das Export-Aktionssymbol den sogenannten Exportmodus von HotHelp aktivieren. In diesem Modus sind weder die beiden Eingabefelder noch irgendwelche Querverweise verwendbar, so daß Sie den angezeigten Text nicht ändern können. Zu diesem Zweck müssen Sie den Modus erst wieder verlassen; dazu steht das Zurück-Symbol zur Verfügung.

Im nächsten Schritt wählen Sie nun bitte den gewünschten Ausschnitt des Textes aus - es ist ja durchaus möglich, daß Sie nur an einem kleinen Teil eines langen Textes interessiert sind. Dazu gibt es drei Möglichkeiten:

- Ziehen: Klicken Sie dazu mit der Maus auf den Anfang oder das Ende des gewünschten Bereiches. Halten Sie die linke Maustaste gedrückt und fahren bis zum anderen Ende des Bereiches. Falls dieses nicht sichtbar ist, können Sie den sichtbaren Textausschnitt wie üblich durch Überschreiten des Fensterrandes in die entsprechende Richtung verschieben. Lassen Sie die Maustaste dann über der gewünschten Zielzeile los.

- Erweiterte Auswahl: Klicken Sie die Anfangs- oder End-Zeile des gewünschten Bereiches an. Lassen Sie dann die linke Maustaste los, drücken eine der beiden Umschalttasten und klicken auf das andere Ende des Bereiches. Der Text zwischen den beiden angewählten Zeilen wird markiert. Da Sie zwischen den beiden Klicks auch die Möglichkeit haben, den sichtbaren Textausschnitt über die Rollsymbole zu verschieben, können Sie so sehr schnell große Blöcke auswählen.

- Doppelklick: Wenn Sie mit der Maus einen Doppelklick auf eine beliebige Zeile vornehmen, wird der gesamte aktuelle Hilfstext markiert. Auf diese Weise können Sie sich vor allem bei größeren Texten das Auf- und Abrollen sparen, das zur Markierung des Textes mit den beiden anderen Methoden notwendig ist.

Der markierte Text kann nun über die vier Aktionssymbole in der linken Hälfte der Symbolleiste exportiert werden.

Das Drucker-Symbol gibt den Text (wer hätte es gedacht) auf einen angeschlossenen Drucker aus. Der Ausdruck erfolgt anhand der Systemvoreinstellungen, die über das entsprechende Systemprogramm (Preferences bzw. Printer) gesteuert werden können.

Über das Datei-Symbol können Sie den markierten Bereich in einer normalen Textdatei abspeichern, die Sie hinterher mit jedem Editor und jeder Textverarbeitung wieder einlesen können. HotHelp öffnet dazu ein Datei-Auswahlfenster, in dem Sie den Namen der Zieldatei eingeben können.

In den Amiga-Zwischenspeicher, das sog. Clipboard, wird der Text über das dritte Symbol kopiert. Das Clipboard wird mittlerweile von vielen Programmen unterstützt, so daß Sie sich den Umweg über eine Datei sparen können.

Das vierte Symbol mit dem schlichten Titel 'Einfügen' schließt das HotHelp-Fenster automatisch. Anschließend wird dann der angewählte Text so behandelt, als hätten Sie ihn selbst über die Tastatur eingegeben. War z.B. vor dem Öffnen des HotHelp-Fensters ein Shell-Fenster aktiv, wird der markierte Text in dieses Fenster geschrieben und die Shell versucht, den Text als Shell-Befehle zu interpretieren - je nach Inhalt mit mehr oder weniger Erfolg. Gehört das zuletzt aktive Fenster hingegen zu einem Textverarbeitungsprogramm, wird der markierte Abschnitt dort in den Text eingetragen. Um den Einfügevorgang abzubrechen, genügt es, eine Maustaste zu drücken.

6.10. Verschiedene Ansichten über HotHelp

Hier soll nicht etwa die Meinung mehrerer zufriedener HotHelp-Anwender vorgestellt werden - statt dessen wird hier gezeigt, wie Sie über HotHelp Ansichten verschiedener Hilfstexte in einem Fenster zusammenfassen können. Zu diesem Zweck existiert das Ansichten-Menü.

Ansichten bieten die Möglichkeit, in einem einzigen HotHelp-Fenster beliebig viele verschiedene Texte gleichzeitig bereitzuhalten - jeder Text befindet sich in seiner eigenen Textansicht. Jede Ansicht verhält sich im Prinzip wie ein vollkommen unabhängiges HotHelp-Fenster. Die Anzahl von Ansichten, die in einem Fenster eingerichtet werden können, wird lediglich durch die aktuelle Größe des Fensters beschränkt; je größer das Fenster ist, desto mehr Ansichten finden auch darin Platz. Über die Maus oder die Tastatur kann einfach zwischen den verschiedenen Ansichten hin- und hergeschaltet werden.

Sie erkennen ein HotHelp-Fenster mit zwei oder mehr Ansichten an den Querbalken im Textbereich des HotHelp-Fensters. Diese sind jeweils mit dem Titel des Textes beschriftet, der sich in der entsprechenden Ansicht befindet. Sie können so auf einen Blick erkennen, welche Texte in den verschiedenen Ansichten geladen sind.

Bis auf eine sind alle diese Ansicht-Titelleisten mit der normalen Hintergrundfarbe gefüllt. Die einzige Ausnahme stellt hierbei die sogenannte aktive Ansicht dar - das ist diejenige, deren Inhalt gerade angezeigt wird. Ihr Titel hat einen farbigen Hintergrund. Alle Aktionen, die Sie durchführen, beziehen sich immer nur auf diese aktive Ansicht; alle anderen Ansichten werden davon nicht beeinflußt. Sie können also mit der aktiven Ansicht genauso arbeiten wie mit einem eigenen HotHelp-Fenster.

Klicken Sie nun mit der Maus auf den Titel einer anderen Ansicht, so wird diese aktiviert, und ihr Inhalt wird im Fenster dargestellt. Die vorher aktive Ansicht schrumpft bis auf ihre Titelleiste zusammen. Sie können sich auch über zwei Menübefehle vorwärts und rückwärts durch die Ansichten bewegen. Ein weiterer Menübefehl schließt und entfernt die aktuelle Ansicht wieder, wobei automatisch ihre Vorgängerin aktiviert wird.

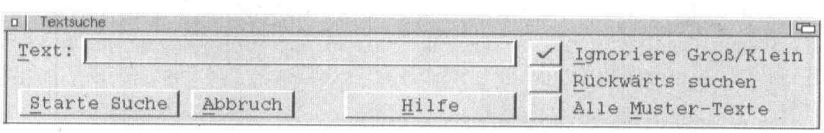
Sehr nützlich ist auch noch die Möglichkeit, eine neue Ansicht zu öffnen, in der automatisch der zum aktuellen Querverweis gehörende Hilfstext dargestellt wird. Zu diesem Zweck kann der unterste Menüpunkt 'Querverweis-Ansicht' verwendet werden.

6.11. Die integrierte Suchfunktion

Das neue Suchen-Menü bietet Ihnen die Möglichkeit, HotHelps Hilfstexte nach beliebigen Worten zu durchsuchen - ganz so, wie Sie es von Ihrem Editor oder Ihrer Textverarbeitung auch gewöhnt sind. Dies ist vor allem bei längeren Hilfstexten recht nützlich.

Die ersten drei Einträge des Suchen-Menüs finden sich so oder ähnlich in den meisten Programmen wieder: 'Suche Text...' öffnet ein Kommunikationsfenster, in dem Sie den Suchtext eingeben und verschiedene Einstellungen vornehmen können; über die beiden nächsten Menüpunkte können Sie dann das vorige oder nächste Auftreten des Suchwortes anzeigen lassen. Das gefundene Wort wird farblich invertiert dargestellt und (um es vom aktuellen Querverweis zu unterscheiden) zusätzlich mit einem Rahmen umgeben. Falls das Wort nicht gefunden wird, läßt HotHelp den Bildschirm kurz aufblitzen.

Der Menüpunkt 'Schlüsselposition' sucht nach dem ersten Auftreten des aktuellen Schlüsselbegriffes im Text und kann bei längeren Texten recht nützlich sein. Der letzte Menüpunkt 'Lösche Markierung' schließlich entfernt die Markierung des zuletzt gefundenen Wortes wieder.



Im Fenster für die Einstellung der Suchoptionen finden Sie ein Texteingabefeld, in das Sie den Suchtext eingeben können. Starten Sie die Suche dann bitte, indem Sie die Eingabe mit der Eingabetaste bestätigen oder das entsprechende Symbol auswählen. Auch die beiden oberen Auswahlfelder tauchen so oder ähnlich in jeder Textverarbeitung auf: Das obere bestimmt, ob bei der Suche die Groß-/Kleinschreibung beachtet werden soll; das darunter liegende steuert die Suchrichtung - Vorwärtssuche in Richtung des Textendes oder Rückwärtssuche auf den Anfang des Textes zu.

Am interessantesten ist jedoch das unterste Auswahlfeld mit dem Titel 'Alle Muster-Texte'. Wird dieses Symbol angewählt, beschränkt sich HotHelp bei der Suche nicht nur auf den aktuellen Text, sondern untersucht der Reihe nach alle Texte, die zum aktuellen Projekt- und Schlüssel-Muster passen. Sie haben damit die Möglichkeit, auch Begriffe, die nicht als Schlüssel definiert wurden, in HotHelp-Texten aufzufinden.

Wollen Sie z.B. alle Texte des Shell-Projektes sehen, in denen das Wort 'Brabbelt' auftaucht, geben Sie als Projektname 'Shell' und als Schlüsselmuster '*' ein. Öffnen Sie dann das Hilfsfenster, geben 'Brabbelt' als Suchtext ein und aktivieren die Suche in allen Texten. HotHelp lädt dann nacheinander alle Texte des Shell-Projektes und durchsucht sie. Natürlich ist diese Art der Suche nicht besonders schnell - dafür wissen Sie anschließend aber auch, welcher der Shell-Befehle zum Brabbeln neigt...

6.12. Verwendung von Lesezeichen

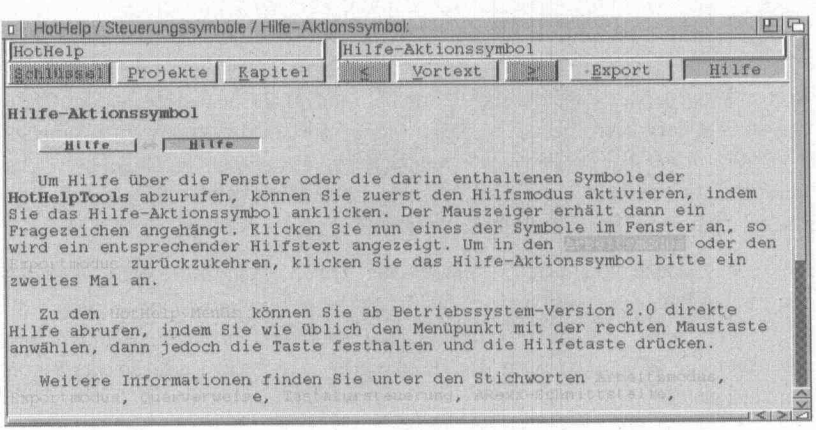
Über das Lesezeichen-Menü haben Sie die Möglichkeit, HotHelp-Texte durch elektronische Eselsohren zu kennzeichnen. Sie können bis zu fünf Marken in beliebige Texte setzen, die Sie dann jederzeit wieder abrufen können (natürlich können Sie nur Marken abrufen, die Sie vorher auch gesetzt haben).

Dies kann recht nützlich sein, wenn Sie mehrmals zwischen einer Reihe von Texten hin- und herspringen müssen. Setzen Sie in jeden der Texte eine Marke, können Sie die Texte ganz einfach wieder abrufen.

Der unterste Menüpunkt dient zum Löschen aller gesetzten Lesezeichen - er bügelt die Eselsohren wieder glatt...

6.13. Hilfe über Hilfe

Bis hierher wurde versucht, Ihnen einen möglichst umfassenden Überblick über die Bedienung von HotHelp-Fenstern zu geben. Falls Ihnen bei dieser "Tour de Force" durch HotHelp jedoch die Bedeutung des einen oder anderen Bedienungselementes wieder entfallen sein sollte, können Sie jederzeit Hilfe dazu abrufen. Zu diesem Zweck steht das Hilfe-Symbol zur Verfügung. Wenn Sie es anwählen, verwandelt der Mauszeiger sich in einen kurzen Pfeil mit einem angehängten Fragezeichen. Klicken Sie nun eines der Symbole auf dem Bildschirm an, zeigt HotHelp einen entsprechenden Hilfstext an, der die genaue Bedeutung dieses Symbols beschreibt.



Wenn Sie über Betriebssystem-Version 2.0 oder höher verfügen, können Sie auch zu den Menüs ganz einfach Hilfe abrufen: Wählen Sie dazu den Menüpunkt wie üblich mit der rechten Maustaste an. Halten Sie dann die Maustaste jedoch weiterhin fest und drücken noch zusätzlich die Hilfetaste auf der Tastatur - HotHelp zeigt Ihnen dann einen Text zu dem entsprechenden Menüpunkt an. Befand der Mauszeiger sich über dem Titel eines Menüs, wird statt dessen ein allgemeiner Text über das Menü selbst angezeigt, ohne auf einen speziellen Menüpunkt einzugehen.

Falls Sie noch nicht mit OS 2.0 arbeiten (warum eigentlich nicht?), sollten Sie sich den Text etwas genauer ansehen, der beim Betätigen des Hilfe-Symbols angezeigt wird: er enthält eine Reihe von Querverweisen, unter anderem auch auf die HotHelp-Menüs. Sie können die Texte über die Menüs dann wie gewohnt über die Querverweise abrufen und durchlesen.

Auch wenn Sie OS 2.0 oder eine höhere Version verwenden, sollten Sie sich diesen Text ruhig genauer ansehen - er enthält noch eine Reihe von Verweisen auf andere, interessante Themen, auf die in dieser Anleitung nicht weiter eingegangen wird.

Der Hilfsmodus bleibt solange aktiv, bis Sie das Hilfe-Symbol nochmals anwählen (oder bis Sie das Fenster wieder schließen).

6.14. Tastaturbedienung

HotHelp kann fast hundertprozentig über die Tastatur gesteuert werden - der Griff zur Maus ist nur in Ausnahmefällen notwendig!

Die Menüs können wie üblich durch Kombination des angezeigten Buchstabens mit der rechten Amiga-Taste angewählt werden (bzw. durch eine Zifferntaste zusammen mit der Controltaste beim Setzen der Lesezeichen).

Für die Symbole stehen bis zu drei Tastatur-Abkürzungen zur Verfügung - der in der Beschriftung unterstrichene Buchstabe, eine Funktionstaste und ggf. eine Taste auf dem Ziffernblock am rechten Rand der Tastatur. Wenn Sie (wie oben beschrieben) einen Hilfstext über eines der Symbole abrufen, zeigt HotHelp dort auch alle Tastaturkürzel des entsprechenden Symbols an.

Außerdem finden Sie im HotHelp-Projekt noch den Schlüssel 'Tastatursteuerung', unter dem sämtliche Tastatur-Bedienungsmöglichkeiten von HotHelp erläutert werden. Unter anderem erfahren Sie dort, wie der Text über die Tastatur verschoben werden kann oder wo Sie die wichtigsten Funktionen auf dem numerischen Tastenfeld wiederfinden. Sie erreichen diesen Text auch durch einen Querverweis aus demjenigen Hilfstext heraus, der bei Betätigung des Hilfe-Symbols dargestellt wird.

6.15. Möglichkeiten, HotHelp aufzurufen

Bisher wurde nur gezeigt, daß HotHelp-Fenster über die Tastenkombination <Alt>-<Help> geöffnet werden können (dies ist eine Abkürzung für 'Halten Sie die Alt-Taste gedrückt und drücken dann zusätzlich die Hilfetaste'). Wenn Sie das Fenster mehrmals schließen und über diese Kombination wieder öffnen, sehen Sie jedesmal denselben Text vor sich.

Es gibt jedoch noch drei andere Möglichkeiten, das HotHelp-Fenster zu öffnen. Dort wird dann nicht der zuletzt gezeigte Text wieder dargestellt; statt dessen bietet jede der drei Tastatur-Kombinationen die Möglichkeit, HotHelp einen bestimmten Schlüssel vorzugeben, zu dem dann Hilfe dargestellt werden soll.

Am interessantesten ist sicherlich die Kombination <Shift>-<Help>. Über sie kann Hilfe zum aktuellen Wort unter dem Cursor eines Editors abgerufen werden. Unterstützt werden die Editoren Edward von MAXON, CygnusEd von CygnusSoft und TurboText von Oxxi (voreingestellt ist der Edward, über das Programm HotHelpPref kann diese Einstellung jedoch bequem geändert werden). Dieser Tastenbefehl ist vor allem für den Programmierer extrem nützlich, da so z.B. zu jedem Begriff des Amiga-Betriebssystems (sei dies eine Funktion, eine Struktur oder eine vordefinierte Konstante) direkt und einfach Hilfe abgerufen werden kann.

Erkennt HotHelp die Tastenkombination <Ctrl>-<Help>, übernimmt das Programm das zuletzt von Ihnen eingegebene Wort und versucht, Hilfe dazu darzustellen. Dabei ist es gleichgültig, ob Sie dieses Wort in einen Editor, eine Textverarbeitung, ein Shell- oder ein HotHelp-Fenster eingegeben haben - HotHelp überwacht grundsätzlich alle Tastatureingaben und hält das jeweils zuletzt eingegebene Wort fest. Sie können dies z.B. verwenden, wenn Sie in der Shell gerade einen Befehl eingegeben haben, sich jedoch nicht mehr an die Parameter erinnern können. Da die letzte Eingabe jedoch der Name des Shell-Befehls war, können Sie über <Ctrl>-<Help> einfach einen dazu passenden Hilfstext abrufen!

Die letzte Tastenkombination <Amiga>-<Help> entnimmt den gesuchten Begriff aus dem Amiga-Zwischenspeicher (Clipboard) und öffnet ein Fenster mit einem dazu passenden Text. Ab Betriebssystemversion 2.0 können Sie diesen Hotkey z.B. verwenden, um Hilfe zu einem Wort in einem Shell-Fenster abzurufen. Markieren Sie dazu das Wort über die Maus, kopieren es über <Rechte Amiga>-<c> in den Zwischenspeicher und rufen dann über <Amiga>-<Help> Hilfe dazu ab - voila!

7. Onlineprojekte

Mit den Onlineprojekten bietet HotHelp eine faszinierende neue Möglichkeit, Texte auf einfachstem Wege in HotHelp zu integrieren. Im Gegensatz zu den normalen Projekten, deren Inhalt fest und unveränderlich ist, können Onlineprojekte jederzeit erweitert und verändert werden. Wenn Sie sich die normalen Projekte als Bücher einer Bibliothek darstellen, dann sind Onlineprojekte leere Notizbücher, in die Sie Texte eintragen, umsortieren, ändern und auch wieder löschen können. Dabei verhalten sich Onlineprojekte ganz genauso wie die normalen Projekte, so daß Sie bei der Abfrage der Texte keinerlei Unterschied feststellen werden.

7.1. Benutzung der Onlineprojekte

Um einen neuen Text zu dem aktuellen Onlineprojekt hinzuzufügen, sind nur wenige Tastendrücke erforderlich. Voraussetzung ist allerdings, daß Sie einen der drei folgenden Editoren verwenden: Edward (MAXON), CygnusEd (CygnusSoft) oder TurboText (Oxxi). Voreingestellt ist der Edward; Sie können jedoch über HotHelpPref jederzeit auch einen anderen Editor anwählen.

Benutzen Sie nun Ihren Editor, um den gewünschten Text zu schreiben. Markieren Sie den Text dann als Block und drücken anschliessend die Tastenkombination <Ctrl>-<Linke Amiga>-<s>. Dadurch wird der gerade markierte Block an HotHelp übergeben.

Sie müssen nun noch einen Schlüssel festlegen, unter dem der Text in das Onlineprojekt eingetragen werden soll, damit Sie ihn auch später wiederfinden. Dazu öffnet HotHelp ein kleines Fenster, in dem Sie sowohl den Schlüsselbegriff als auch den Namen des Onlineprojektes, in das der Schlüssel eingetragen werden soll, vorgeben können. Falls Sie mehrere Onlineprojekte verwenden, können Sie über die beiden Doppelpfeil-Symbole neben dem Projekt-Eingabefeld alle Onlineprojekte der Reihe nach durchgehen. Geben Sie einen neuen Projektnamen an, so wird das entsprechende Onlineprojekt neu angelegt.

Sind Sie mit den Einstellungen zufrieden, können Sie den neuen Schlüssel mit dem Übernehmen-Symbol in das gewählte Onlineprojekt eintragen. Wenn Sie jetzt im HotHelp-Fenster die Projektübersicht abrufen, sehen Sie dort ein weiteres Projekt mit dem Namen Ihres Onlineprojektes. Wählen Sie dieses aus, zeigt HotHelp Ihnen eine Übersicht mit allen Schlüsseln dieses Projektes an, aus der Sie dann wie üblich einen auswählen können.

Auch das Ändern eines so erstellten Textes ist ohne weiteres möglich: öffnen Sie dazu das HotHelp-Fenster und stellen Sie den gewünschten Text dar. Exportieren Sie ihn dann in Ihren Editor (eine ausführliche Anleitung zum Export von Hilfstexten finden Sie weiter oben). Sie können dazu z.B. das Clipboard- oder das Einfügen-Symbol verwenden.

Befindet der Text sich nun im Editor, können Sie ihn nach Herzenslust ändern. Anschließend können Sie ihn dann wieder (wie oben beschrieben) unter dem entsprechenden Schlüsselbegriff in das Onlineprojekt übernehmen (HotHelp ist dann jedoch mißtrauisch und fragt sicherheitshalber nach,

ob der alte Text tatsächlich durch den neuen ersetzt werden soll; beruhigen Sie es einfach, indem Sie hier mit 'Ja' antworten...).

7.2. Anwendungsgebiete für Onlineprojekte

Onlineprojekte können praktisch überall eingesetzt werden - lassen Sie Ihrer Phantasie also ruhig freien Lauf. Grundsätzlich ist ein Onlineprojekt überall da sinnvoll, wo kürzere Texte schnell festgehalten und später wieder über HotHelp abgerufen werden sollen, wobei die Texte auch möglichst leicht zu ändern sein sollen und wo auf weiter verzweigende Querverweise innerhalb der Hilfstexte verzichtet werden kann. Hier einige (mehr oder weniger) sinnvolle Vorschläge:

Programmierer können alle Funktionsprototypen und Strukturvereinbarungen in einem Onlineprojekt ablegen, so daß Sie in Zukunft auch zu jeder selbstgeschriebenen Funktion Hilfe über <Shift>-<Help> abrufen können. Ändert sich eine Funktion oder Struktur, wird der entsprechende Schlüssel einfach neu definiert. Über HotHelpPref können Sie auch die automatische Auswahl des Schlüsselbegriffs aus dem Textblock einstellen, die HotHelp ein gewisses Maß an Intelligenz verleiht und hierbei sehr hilfreich ist. Da Onlineprojekte auch keiner zahlenmäßigen Begrenzung unterliegen, können Sie sich zu jedem Programm, an dem Sie arbeiten, ein eigenes Onlineprojekt mit den Hilfstexten zu diesem Programm erstellen.

Erstellen Sie sich ein Projekt mit den Adressen aller Freunde und Bekannten, wobei Sie jeweils den Namen als Schlüsselbegriff verwenden. Änderungen (z.B. der Adresse oder der Postleitzahlen...) können Sie dann wie oben beschrieben ohne Probleme durchführen. Wenn Sie einem Bekannten einen Brief schreiben wollen, übernehmen Sie seine Adresse durch eine der zahlreichen Exportmöglichkeiten aus HotHelp in Ihre Textverarbeitung.

7.3. Onlineprojekte und andere Editoren

Auch falls Sie einen anderen als die drei genannten Editoren verwenden, brauchen Sie nicht auf die Verwendung von Onlineprojekten zu verzichten. Voraussetzung ist jedoch, daß Ihr Editor ARexx-fähig ist (s.u.). Sie müssen nur ein ARexx-Skript erstellen, das den gewünschten Textblock in das Clipboard überträgt (falls Ihr Editor das Clipboard nicht sowieso schon zum Speichern von Blöcken verwendet) und dann das Kommando HH_OnlineBlock mit entsprechenden Parametern an den Host HOTHELP_REXX sendet (weitere Informationen finden Sie im nächsten Kapitel).

8. ARexx

ARexx ist seit Version 2.0 fester Bestandteil des Amiga-Betriebssystems. Es handelt sich um eine Basic-ähnliche Sprache, deren besondere Qualität darin liegt, daß mit ihr sehr einfach andere Programme gesteuert werden können. Um mit ARexx zusammenzuarbeiten, muß ein Programm über eine sogenannte ARexx-Schnittstelle verfügen. Wie Sie sich jetzt sicher schon denken können, verfügt auch HotHelp in der neuen Version über eine solche Schnittstelle. Über diese ist es möglich, von ARexx (und damit auch von allen ARexx-fähigen Programmen aus) HotHelp-Fenster zu öffnen, zu steuern und ggf. auch wieder zu schließen.

ARexx-Interessenten finden eine komplette Übersicht über alle ARexx-Befehle von HotHelp im HotHelp-Projekt unter dem Schlüssel 'ARexx-Schnittstelle'.

Außerdem enthält HotHelp jetzt noch ein Projekt mit dem Namen 'ARexx', in dem alle Standard-Befehle und -Funktionen von ARexx erläutert werden.

9. Änderungen seit Version 2

Dieses Kapitel ist sicher für alle interessant, die HotHelp schon von der Version 2 her kennen. Es beschreibt kurz alle Änderungen, die in der neuen Version vorgenommen wurden. Eine ausführlichere Beschreibung finden Sie im vorhergehenden Kapitel oder in HotHelp selbst, wo Sie zu jedem Bedienungselement Hilfe abrufen können.

Wenn Sie Ihr erstes HotHelp-3-Fenster öffnen, wird Ihnen zuerst keinerlei Änderung auffallen. Alle Symbole befinden sich noch am selben Platz wie bisher und haben auch dieselbe Bedeutung - die einzige Ausnahme stellen die beiden Pfeilsymbole dar. Sie wurden bisher dazu verwendet, bei Eingabe eines Musterbegriffes alle passenden Schlüsselbegriffe der Reihe nach anzuzeigen. Da davon jedoch nur relativ selten Gebrauch gemacht wird, werden die Symbole in Version 3 zum Blättern durch ein Projekt verwendet - eine neue Funktion, die unter älteren Versionen noch nicht zur Verfügung stand. Die Funktionen zum Anzeigen aller zu einem Muster passenden Schlüssel wurden in das Schlüssel-Menü verschoben.

Damit sind wir auch schon beim Stichwort 'Menü' angelangt. Jedes HotHelp-Fenster verfügt jetzt über eine Menüleiste, und Sie finden hier einige der interessantesten Neuerungen von HotHelp 3: die Darstellung mehrerer Texte in einem HotHelp-Fenster (Ansichten), die leistungsstarke Suchfunktion (die auch in mehreren Texten nach einem Nicht-Schlüsselbegriff suchen kann) und die Vergabe von Lesezeichen. Außerdem können Sie von hier aus sämtliche HotHelpTools starten.

Die bisher eingebaute Hilfefunktion zu den Bedienungselementen über die rechte Maustaste war damit natürlich hinfällig. In Version 3 können Sie jedoch über das Hilfe-Symbol den Hilfsmodus aktivieren, in dem Sie durch Anklicken mit der linken Maustaste Hilfe zu den Symbolen abrufen können.

Texte können jetzt in beliebige Dateien exportiert werden, die über ein Datei-Auswahlfenster angegeben werden können. Die Beschriftung des Spezial-Symbols zeigt die aktuelle Funktion des Symbols an ('Editor' oder 'Einfügen').

Alle Symbole können jetzt (wie von Commodore empfohlen) durch Eingabe des in der Symbolbeschriftung unterstrichenen Buchstabens über die Tastatur ausgelöst werden - die bisherige, nicht standard-gemäße Kombination mit einer der Amigatasten ist nicht mehr notwendig und auch nicht mehr möglich. Bei der Auswahl eines Querverweises durch Eingabe seiner Anfangsbuchstaben muß dafür jetzt gleichzeitig eine der Alt-Tasten gedrückt werden.

Bei der Eingabe eines Musters in die beiden Eingabefelder am oberen Rand des Fensters können nun auch die aus der Shell bekannten Musterzeichen verwendet werden - vorausgesetzt, Sie arbeiten mit OS 2.0 oder höher.

Der angezeigte Hilfstext kann nun auch mit der Maus gerollt werden, indem diese bei festgehaltener linker Maustaste aus dem Textbereich herausgezogen wird.

Über die Tastenkombination <Amiga>-<Help> kann Hilfe zum aktuellen Inhalt des Zwischenspeichers (Clipboard) abgerufen werden.

HotHelp arbeitet nun auch bei <Shift>-<Help> mit dem neuen CygnusEd V3.5 zusammen - aufgrund einer inkompatiblen Änderung der ARexx-Schnittstelle dieses Editors hatte HotHelp 2 damit leider so einige Probleme...

Die Tastenkombination <Ctrl>-<Linke Amiga>-<s> übernimmt den aktuellen Block Ihres Editors und speichert ihn im aktuellen Onlineprojekt ab, wo er direkt jedem HotHelp-Fenster zur Verfügung steht. Mehr über Onlineprojekte finden Sie in Kapitel 7.

Komplett überarbeitet wurde das Programm HotHelpManager mit seiner manchmal doch etwas seltsam anmutenden Benutzerführung. Die drei Module 'Projektmanager', 'Voreinstellungen' und 'Projekte übersetzen' wurden herausgelöst und liegen nun als eigenständige Programme vor (HotHelpPro, HotHelpPref und HotHelpComp). Dazu kommen noch EasyHotHelp und der HotHelpMarker, die die Erstellung eigener Projekte extrem vereinfachen. In einer editor-ähnlichen Oberfläche können alle erforderlichen Kennungen über die Maus oder die Tastatur in die Quelltexte eingefügt werden; auch die Übersetzung, die Anzeige von Übersetzungsfehlern und der Test neuer Projekte wird mit ihm zum Kinderspiel.

Alle Programme wurden völlig neu entwickelt, wobei auch die (oftmals recht verzwickte) Benutzerführung komplett überarbeitet wurde. Vergessen Sie also am besten alles, was Sie über den alten HotHelpManager wissen und machen Sie sich mit den neuen Tools vertraut!

Falls Sie schon unter HotHelp 2 Projekte erstellt haben, hier noch einige Hinweise:

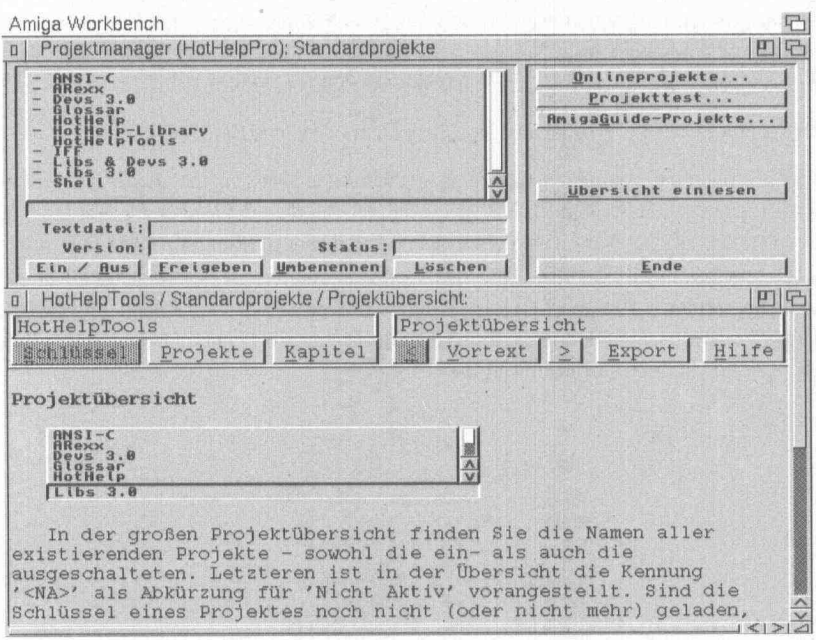
Unter HotHelp 2 konnten die einzelnen Marken bis zu 99 Zeichen lang sein; dies wurde in HotHelp 3 auf 49 Zeichen begrenzt. Außerdem (da das Suchverfahren, mit dem die Marken im Text gesucht werden, komplett überarbeitet wurde) sind überlappende Marken jetzt grundsätzlich verboten (wie z.B. 'Schlüssel' und 'SchlüsselUnsichtbar' - die erste Marke ist komplett in der zweiten enthalten). HotHelpComp überprüft alle angegebenen Marken und macht Sie ggf. vor dem Beginn der Übersetzung auf Fehler dieser Art aufmerksam. Sie müssen dann die entsprechenden Marken in den Projekt-Einstellungen und in Ihren Quelltexten anpassen.

10. Die HotHelpTools

Bisher haben Sie gesehen, wie Sie HotHelp-Fenster aufrufen und steuern können. Im folgenden wird nun auf die HotHelpTools eingegangen - dabei handelt es sich um eine Reihe von Programmen, die verschiedene Aufgaben rund um HotHelp herum übernehmen (Voreinstellungen, Wartung der Standard-, AmigaGuide- und Online-Projektdateien, Erstellung eigener Projekte).

10.1. Die Hilfsfunktion der HotHelpTools

Alle Tools verfügen über eine eingebaute, kontextsensitive Online-Hilfe - natürlich auf Basis des HotHelp-Systems. Sie gibt Ihnen Auskunft über die einzelnen Fenster der Tools, die darin auftretenden Steuerungs-Symbole und die Menüs. Sie können diese Hilfe abrufen, indem Sie den Hilfe-Menüpunkt anwählen oder (noch einfacher) die Hilfetaste drücken. Daraufhin wird ein HotHelp-Fenster geöffnet, in dem Sie einen Text über das entsprechende Tool und das gerade aktive Fenster finden. Sie können von hier aus nun über Querverweise weitere Informationen abrufen.



Noch einfacher ist es allerdings, Hilfe zu bestimmten Symbolen oder Bereichen im Fenster des Tools abzufragen. Der Mauszeiger verändert sein Aussehen, sobald Sie den Hilfsmodus aktivieren - er bekommt ein Fragezeichen angehängt. Wenn Sie mit diesem Fragezeichenzeiger auf ein Symbol im Fenster eines HotHelpTools klicken, wird in dem vorher geöffneten HotHelp-Fenster ein Hilfstext angezeigt, der die Bedeutung des angeklickten Elementes erläutert. Sie können auf diese Weise Hilfe zu allen sichtbaren Symbolen abrufen.

Um die Arbeit mit dem Tool fortzusetzen, müssen Sie zuvor den Hilfsmodus beenden. Dazu existieren zwei Möglichkeiten: Schließen Sie das HotHelp-Fenster mit den Hilfstexten oder aktivieren Sie das Fenster, von dem aus Sie den Hilfsmodus aufgerufen haben, und drücken dort nochmals die Hilfetaste. Die letztgenannte Methode hat den Vorteil, daß das HotHelp-Fenster dann geöffnet bleibt, Sie den angezeigten Hilfstext also auch weiterhin vor Augen haben.

Im den Programmen HotHelpMarker und EasyHotHelp wurde die Hilfsfunktion etwas anders gelöst, da den einzelnen Symbolen dort keine so große Bedeutung zukommt wie in den übrigen Programmen. Im Hilfsmodus wird Ihnen ein Text über das aktuelle Fenster angezeigt. Sie finden hier bereits eine Erklärung aller wichtigen Symbole vor. Auf die Möglichkeit der Auswahl eines speziellen Symbols wurde daher verzichtet.

Da in diesen beiden Programmen jedoch den Menüs (der HotHelpMarker allein bietet schon fast 100 Menüpunkte an) eine sehr wichtige Bedeutung zukommt, können Sie ab Betriebssystemversion 2.0 direkt Hilfe zu jedem Menüpunkt abrufen. Wählen Sie dazu den Menüpunkt wie üblich an. Anstatt die Maustaste dann loszulassen, drücken Sie jedoch noch zusätzlich die Hilfetaste. Das Programm öffnet dann ein HotHelp-Fenster mit einem Hilfstext über den Menüpunkt. Befindet der Mauszeiger sich über dem Menütitel oder außerhalb des Menüs, wenn die Help-Taste gedrückt wird, sehen Sie einen Hilfstext über das gesamte Menü.

Diese ausführliche Online-Hilfe ist auch der Grund dafür, daß in der folgenden Anleitung nicht auf jedes einzelne Symbol und jeden Menüpunkt eingegangen wird. Langweilige Erklärungen der Form "Um dies und das einzustellen, wählen Sie aus dem XYZ-Menü den Eintrag ABC; im sich daraufhin öffnenden Fenster klicken Sie mit der linken Maustaste auf den 0815-Schieberegler und ziehen ihn bei gedrückt gehaltener Maustaste auf die gewünschte Position. Schließen Sie das Fenster über das Verwenden-Symbol" werden Sie hier also umsonst suchen (da alleine schon das Programm HotHelpPref an die 200 Symbole beinhaltet, würde die Anleitung andernfalls bald den Umfang des neuen Postleidzahlenbuches erreichen...). Vielmehr wird im folgenden darauf eingegangen, WAS Sie mit den Programmen anfangen können - wie Sie die jeweilige Aktion dann genau durchführen, können Sie jederzeit in den Hilfstexten nachlesen!

10.2. Start der HotHelpTools

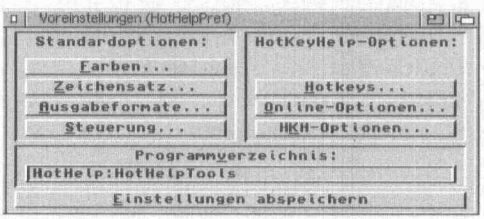
Die Programme können wie üblich von der Workbench oder der Shell aus gestartet werden. Shell-Benutzer können bei einigen Programmen außerdem beim Start zusätzliche Parameter angeben; eine Übersicht darüber erhalten Sie, wenn Sie das jeweilige Programm mit einem Fragezeichen als einzigem Parameter aufrufen. Geben Sie als Parameter 'HILFE' an, erhalten Sie einen ausführlicheren Hinweistext.

Außerdem können alle Tools aus dem HotHelpTools-Menü gestartet werden, das Sie in den Menüleisten der anderen Tools und des HotHelp-Fensters finden.

10.3. HotHelpPref - HotHelps Voreinstellungen

Wenn Sie schon mit Betriebssystem-Version 2.0 oder höher arbeiten, ist Ihnen das Prinzip der verschiedenen Voreinstellungs-Editoren sicher schon von der Workbench vertraut. Bei HotHelpPref handelt es sich ebenfalls um einen solchen Editor, mit dem Sie eine ganze Reihe (knapp 80) von verschiedenen Einstellungen des HotHelp-Systems steuern können.

Nach dem Start sehen Sie zuerst das Hauptfenster des Programms, von dem aus Sie sieben weitere Fenster öffnen können. Jedes dieser Fenster enthält Einstellungen zu einem ganz bestimmten Bereich. Die vier Symbole in der linken Spalte sind dabei für Optionen zuständig, die für alle HotHelp-Fenster gelten (also z.B. auch die, die von den Tools im Hilfsmodus geöffnet werden); die drei rechten Symbole steuern Optionen, die nur für das Programm HotKeyHelp gültig sind (dies ist das Programm, über das HotHelp-Fenster auf Tastendruck geöffnet werden können und das die Online-Projekte und die ARexx-Schnittstelle unterstützt).



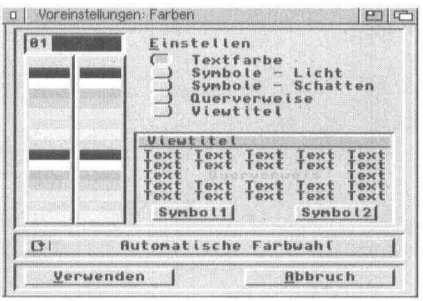
Außerdem sehen Sie hier noch ein Eingabefeld, das nur in Ausnahmefällen interessant ist (es enthält den Namen des Verzeichnisses, in dem HotHelp die HotHelpTools erwartet), sowie ein Symbol zum Abspeichern der Voreinstellungen. Nur, wenn die geänderten Einstellungen über dieses Symbol gesichert werden, werden sie von HotHelp auch verwendet.

Falls Ihr Betriebssystem noch keine verschiedenen Sprachen unterstützt, sehen Sie hier außerdem noch ein Symbol, mit dem Sie die Sprache von HotHelps Benutzerführung regeln können. Ansonsten verwendet HotHelp die eingestellte Systemsprache.

Im weiteren werden nun die Einstellmöglichkeiten, die in den verschiedenen Fenstern vorgenommen werden können, kurz erläutert. In jedem Fenster finden Sie am unteren Rand ein Verwenden- und ein Abbruch-Symbol. Nur wenn das Fenster über das Verwenden-Symbol geschlossen wird, werden die von Ihnen vorgenommenen Änderungen tatsächlich übernommen. Benutzen Sie das Abbruch-Symbol oder schließen das Fenster über das Schließsymbol, setzt HotHelpPref die Änderungen wieder zurück auf den Zustand vor dem Öffnen des jeweiligen Fensters.

10.3.1. Fenster 1: Farben

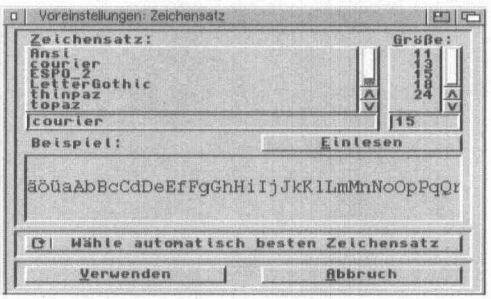
Über das Blättersymbol am unteren Fensterrand können Sie festlegen, ob HotHelp die darüber eingestellten Farben grundsätzlich verwenden soll oder ob die HotHelp-Fenster sich automatisch an die Farbzusammenstellung des verwendeten Bildschirms anpassen sollen. Normalerweise ist die zweite Methode vorzuziehen.



Falls Sie sich für die Vorgabe von Farben entscheiden, können Sie im darüberliegenden Bereich Farben für den Text, die helle und die dunkle Kante der Symbole, die Querverweise und den Titel der aktiven Ansicht angeben.

10.3.2. Fenster 2: Zeichensatz

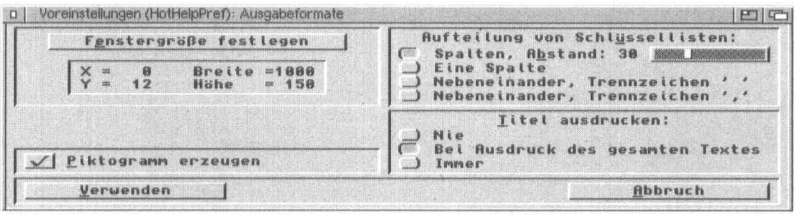
Ähnlich wie im Farbenfenster können Sie auch hier über ein Blättersymbol einen festen Zeichensatz vorgeben oder HotHelp anweisen, den Zeichensatz automatisch an die aktuelle Bildschirmauflösung anzupassen. Daneben können Sie auch noch festlegen, daß beim Öffnen immer der Zeichensatz des zuletzt aktiven Fensters übernommen werden soll. Wenn Sie sich bei der Installation für eine Version mit beschleunigter Textausgabe entschieden haben, steht Ihnen noch eine weitere entsprechende Option zur Verfügung.



Im Bereich über dem Blättersymbol können Sie einen Zeichensatz auswählen. Je nach Einstellung wird dieser grundsätzlich verwendet oder nur in die automatische Entscheidung mit einbezogen. Außerdem sehen Sie immer eine Schriftprobe des aktuellen Zeichensatzes.

10.3.3. Fenster 3: Ausgabeformate

In diesem Fenster können Sie alle Optionen editieren, die sich irgendwie mit der Ausgabe von Hilfstexten befassen.



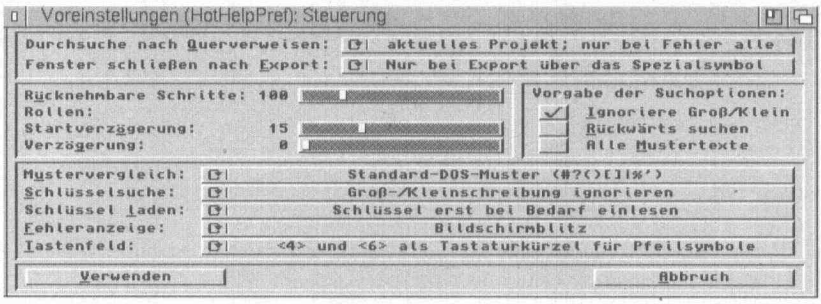
Dazu zählt als erstes die Größe und Position des Fensters. Diese Werte werden immer verwendet, wenn Sie nach einem Neustart des Rechners das erste HotHelp-Fenster öffnen.

Als zweites können Sie Einfluß auf die Aufteilung von Schlüssellisten nehmen. Es handelt sich dabei um Übersichten von Schlüsseln, die HotHelp z.B. bei Eingabe eines Musters erzeugt. Auch in normalen Hilfstexten können solche Listen auftreten. Diese Einstellung ist letzten Endes ein reine Geschmacksfrage.

Die darunter liegenden Optionen befassen sich mit dem Export von Hilfstexten. Sie können dort angeben, ob beim Export in eine Datei ein Workbench-Piktogramm erzeugt werden soll (reine Shell-Benutzer werden diese Option sicherlich gerne ausschalten) und unter welchen Bedingungen dem Ausdruck eines Hilfstextes sein Titel vorangestellt werden soll.

10.3.4. Fenster 4: Steuerung

In diesem Fenster mit seinem allgemeinen Namen wurden letztlich alle Optionen untergebracht, die sich keinem anderen Fenster mehr zuordnen ließen.



Im obersten Feld können Sie den Bereich festlegen, in dem die Suche nach Querverweisen durchgeführt werden soll - entweder nur im aktuellen Projekt, nur zuerst im aktuellen Projekt oder grundsätzlich in allen Projekten.

Darunter wird bestimmt, in welchen Fällen das HotHelp-Fenster nach dem Export geschlossen werden soll.

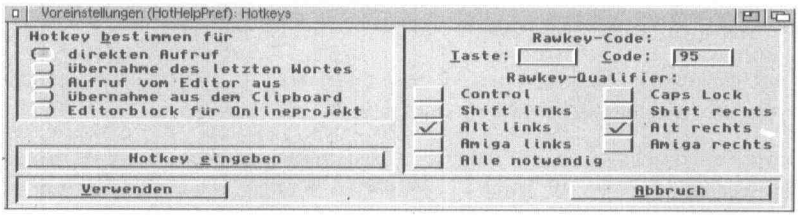
Über die drei Schieberegler können Sie die Anzahl der über das Vortext-Symbol rücknehmbaren Schritte und die Verzögerung beim Rollen über die Maus oder die Rollsymbole im Fensterrahmen einstellen.

Im daneben liegenden Feld können Sie die Suchoptionen festlegen, die beim ersten Öffnen eines HotHelp-Fensters eingestellt sein sollen.

Über die fünf Blättersymbole am unteren Fensterrand können Sie festlegen, ob Sie in den Eingabefeldern die normalen Shell-Muster (erst verfügbar ab Betriebssystemversion 2.0) oder die einfachen HotHelp-Muster verwenden wollen, ob die Groß-/Kleinschreibung eines Begriffs bei der Schlüsselsuche beachtet werden soll, ob die Schlüsselübersichten erst bei Bedarf nachgeladen und bei Speichermangel wieder freigegeben werden dürfen, ob kleine Bedienungsfehler mit einem Bildschirmblitz, einem Piepton, beidem oder nichts von beidem quittiert werden sollen und wie die Tasten 4 und 6 auf dem numerischen Tastenblock verwendet werden sollen.

10.3.5. Fenster 5: Hotkeys

In diesem Fenster können Sie die fünf Tastenkombinationen editieren, die als Hotkeys abgefragt werden sollen. Dies ist sehr sinnvoll, wenn sich eine der HotHelp-Tastenkombinationen mit der eines anderen Programms überschneidet. Sie können hier die entsprechende Kombination ändern.

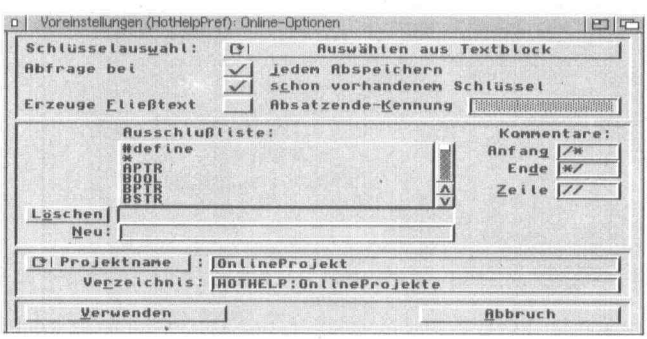


Über die fünf Druckknopfsymbole wählen Sie zuerst aus, welchen der Hotkeys Sie editieren wollen. Im großen Feld rechts können Sie dann den Rawkey-Code und die Qualifier-Kombination einstellen, die für den Hotkey gelten soll. Einfacher ist es allerdings, über das Symbol 'Hotkey eingeben' ein weiteres Fenster zu öffnen, in dem Sie dann die gewünschte Tastenkombination direkt eingeben können, die als neuer Hotkey verwendet werden soll.

10.3.6. Fenster 6: Online-Optionen

In diesem Fenster sind alle Optionen zusammengefaßt, die sich auf Onlineprojekte beziehen.

Im oberen Feld können Sie zuerst die Art und Weise festlegen, auf die der Schlüssel bestimmt werden kann, unter dem ein neuer Text eines Onlineprojektes abgelegt werden soll. Sie können sich entscheiden zwischen Eingabe per Hand, Auswahl aus dem selektierten Block oder automatischer Auswahl.



Bei der letzten Option wird die Ausschlußliste verwendet, die Sie darunter editieren können. HotHelp untersucht dann den Textblock Wort für Wort und benutzt das erste Wort als Schlüssel, das nicht in der Ausschlußliste auftritt. Diese Option ist sehr sinnvoll, wenn Sie z.B. Funktionsprototypen in Online-Projekte aufnehmen (voreingestellt sind die Worte, die bei C-Programmen über-

sprungen werden sollen - Sie können die Liste jedoch jederzeit an Ihre Programmiersprache anpassen).

Weiterhin können Sie festlegen, ob eine Sicherheitsabfrage bei jedem Anlegen eines neuen Textes bzw. vor dem Überschreiben eines schon vorhandenen Textes durchgeführt werden soll. Außerdem kann für den Fall, daß die Onlineprojekte als Fließtext übersetzt werden sollen, eine Marke zur Kennzeichnung von Absatzenden definiert werden.

Neben der Ausschlußliste können Sie auch das Format von Kommentaren angeben. Worte in Kommentaren werden bei der automatischen Schlüsselauswahl nicht berücksichtigt.

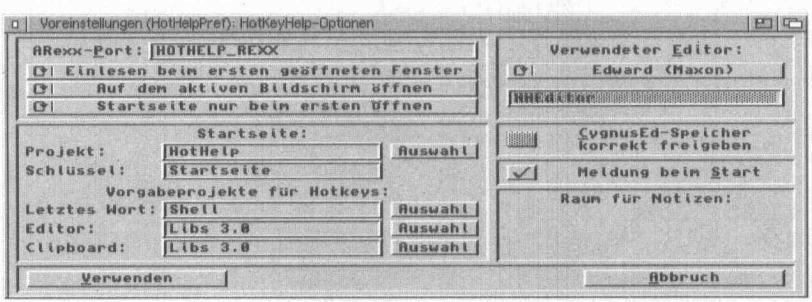
Darunter können Sie schließlich noch den Namen des Onlineprojektes eingeben, in das neue Texte vorgabemäßig eingetragen werden. Ab Betriebssystemversion 2.0 ist auch die Angabe des Namens einer Umgebungsvariablen möglich; HotHelp entnimmt dann beim Einfügen eines neuen Textes den Namen des gewünschten Onlineprojektes aus dieser Variablen.

Das Verzeichnis, in dem Onlineprojekte abgelegt werden, kann ebenfalls hier verändert werden, auch wenn dies im allgemeinen weder notwendig noch ratsam ist.

10.3.7. Fenster 7: HKH-Optionen

Ähnlich wie im Steuerungsfenster sind hier alle übrigen HotKeyHelp-Optionen (HKH-Optionen) zusammengefaßt, die sich jeglicher Zuordnung zu anderen Fenstern entzogen...

Im ersten Feld können Sie den ARexx-Namen festlegen, unter dem HotHelp angesprochen werden kann. Darunter sehen Sie drei Blättersymbole, mit denen Sie bestimmen können, zu welchem Zeitpunkt HotHelp seine Daten einliest, ob das Fenster sich nur auf Public-Bildschirmen öffnen darf und wann die eingestellte Startseite angezeigt werden soll.



Im darunterliegenden Bereich können Sie die Startseite vorgeben, die beim ersten (oder jedem) Öffnen eines Fenster über die Tastenkombination <Alt>-<Help> angezeigt werden soll. Darunter können Sie Vorgabeprojekte für die drei übrigen Hotkeys angeben. Wird der entsprechende Hotkey erkannt, sucht HotHelp den gesuchten Begriff zuerst in dem hier eingestellten Vorgabeprojekt. Über das Auswahl-Symbol können Sie ein Fenster mit einer Projektübersicht öffnen, aus der Sie das gewünschte Projekt auswählen können.

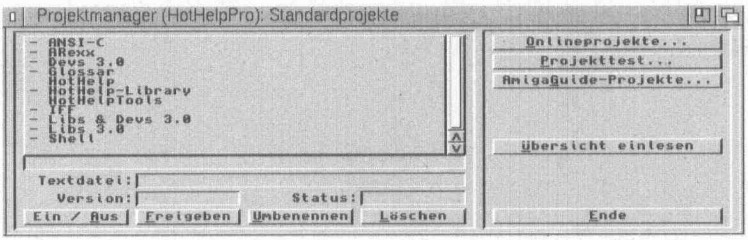
Wenn Sie über <Shift>-<Help> Hilfe zum aktuellen Wort Ihres Editors abrufen oder die Online-Projekte verwenden wollen, ist es notwendig, im Eingabefeld oben rechts den von Ihnen verwendeten Editor einzustellen. Ab Betriebssystemversion 2.0 können Sie auch den Namen einer Umgebungsvariablen angeben, aus der HotHelp bei jeder Verwendung den Namen des aktuellen Editors entnehmen soll.

Das Feld darunter ist nur für Benutzer des CygnusEd vor Version 3.5 interessant, da bei diesen Versionen Speicherverluste bei der Kommunikation mit HotHelp auftreten. Ist das Auswahlfeld aktiviert, korrigiert HotHelp diese Fehler automatisch.

Das letzte Feld schließlich bestimmt, ob HotKeyHelp beim Start kurz ein Fenster mit einer Nachricht öffnen soll oder nicht.

10.4. HotHelpPro - Der Projektmanager

HotHelps Projektmanager enthält vier Module, die alle für die Arbeit mit den verschiedenen Arten von Projektdateien zuständig sind: die Pflege der Standardprojekte (Ein-/Ausschalten, Freigeben von Speicher, Verschieben, Löschen), die Verwaltung der Onlineprojekte, An- und Abmelden von AmigaGuide-Dateien als HotHelp-Projekte sowie die Überprüfung von selbsterstellten Projekten auf Vollständigkeit und logische Fehler. Jedes dieser Module verfügt über ein eigenes Fenster, das aus jedem der anderen Module heraus angesprochen werden kann. Beim Start wird üblicherweise das Standardprojekte-Fenster geöffnet.



10.4.1. Standardprojekte

Den größten Teil dieses Fensters nimmt eine Liste ein, in der Sie die Namen aller vorhandenen Standardprojekte sehen können. Wählen Sie eines der Projekte durch Anklicken aus, zeigt das Programm Ihnen in den Feldern unter der Liste einige Informationen über das Projekt an. Sie sehen dort den Namen der Textdatei des Projektes, seine Versionsnummer und eine Angabe, ob das Projekt ein- oder ausgeschaltet ist.

Über die vier Symbole unterhalb der Liste können Sie nun das ausgewählte Projekt beeinflussen. Das erste Symbol schaltet das Projekt ein bzw. aus. Dies ist sehr nützlich, wenn Sie ein Projekt längere Zeit nicht benötigen. Sie können es dann hier ausschalten, so daß es keinen Speicherplatz verbraucht und auch in der Projektübersicht nicht mehr aufgeführt wird. Wenn Sie dann wieder auf das Projekt zugreifen wollen, können Sie es hier wieder einschalten. Ausgeschalteten Projekten wird in der Liste das Zeichen '0' vorangestellt.

Das 'Freigeben'-Symbol gibt den Speicher frei, den das entsprechende Projekt für die Übersicht über seine Schlüssel angelegt hat, ohne jedoch das gesamte Projekt aus dem Speicher zu entfernen. Sie erkennen dies daran, daß dem Projekt in der Übersicht ein Minuszeichen vorangestellt wird.

Sind Sie sich sicher, daß Sie ein Projekt nie wieder benötigen, können Sie es über das entsprechende Symbol auch komplett löschen. Aber Vorsicht: Einmal gelöschte Dateien sind für immer verloren!

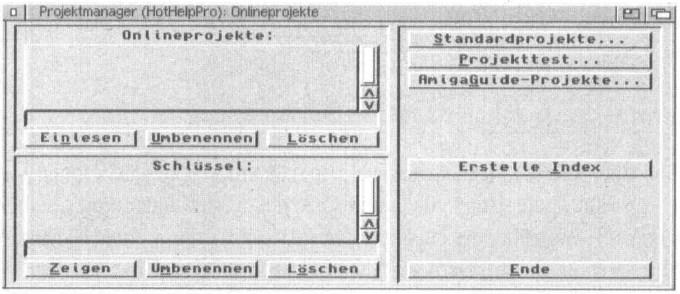
Mit dem Symbol 'Umbenennen' können Sie die Textdatei des Projektes umbenennen oder in ein anderes Verzeichnis verschieben. Dazu eine kleine Erklärung: HotHelps Projekte bestehen aus zwei Dateien; die Kopfdatei mit der Endung ".hdr" enthält Angaben über die Schlüssel des Projektes und

muß sich immer im Verzeichnis "HOTHELP:Projekte" befinden. Die Textdatei mit der Endung ".txt" enthält alle Hilfstexte des Projektes und kann in einem beliebigen Verzeichnis liegen. Sie dürfen die Textdatei allerdings keinesfalls einfach über die Workbench oder die Shell verschieben oder umbenennen, sondern Sie müssen dazu unbedingt dieses Symbol des Projektmanagers benutzen, damit HotHelp nachvollziehen kann, wo sich die Datei befindet.

Über die vier Symbole am rechten Fensterrand können Sie die anderen drei Module abrufen oder das Fenster schließen.

10.4.2. Onlineprojekte

In diesem Fenster können Sie die Texte aller definierten Onlineprojekte bearbeiten. Sie haben die Möglichkeit, ganze Onlineprojekte oder auch einzelne Schlüssel daraus umzubenennen oder ganz zu löschen.

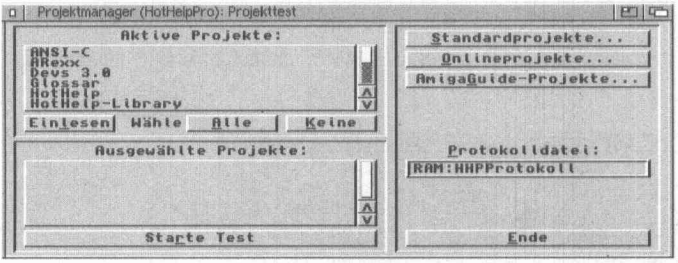


Oben links sehen Sie dazu eine Liste aller Onlineprojekte, aus der Sie das gewünschte Projekt auswählen können. Daraufhin wird in der unteren Liste eine Übersicht über alle Schlüssel dieses Projektes angezeigt. Über das obere Umbenennen- und Löschen-Symbol können Sie nun das ausgewählte Onlineprojekt bearbeiten; wollen Sie nur einen Schlüssel des Onlineprojektes manipulieren, können Sie diesen aus der unteren Liste auswählen und dann umbenennen oder löschen. Um sicherzugehen, können Sie sich den zum angewählten Schlüssel gehörenden Text auch zuvor in einem HotHelp-Fenster anzeigen lassen.

Das Symbol 'Erstelle Index' ist nur für Notfälle vorgesehen, wenn durch einen Diskettenfehler die Indexdatei der Onlineprojekte zerstört wurde (HotHelp hält in dieser Datei Verwaltungsinformationen über alle Onlineprojekte fest). In einem solchen Fall können Sie über dieses Symbol die Indexdatei neu erstellen lassen.

10.4.3. Projekttest

Dieses Modul ist dann für Sie interessant, wenn Sie selbst HotHelp-Projekte erstellen möchten. Es prüft ein oder mehrere Projekte auf logische Fehler. Dazu untersucht das Programm alle definierten Schlüssel und Querverweise des Projektes. Falls zu einem Querverweis ein passender Schlüssel nur in einem anderen Projekt auftaucht oder wenn gar kein passender Schlüssel existiert, zeigt das Programm eine Warnung an. Weiterhin wird noch eine Meldung erzeugt, wenn in einem Projekt ein Schlüssel mehrfach definiert wird - auch dies kann auf einen Fehler hindeuten. Auf diese Weise lassen sich recht einfach schwerwiegende Fehler bei der Erstellung eines Projektes abfangen (z.B. durch falsch geschriebene Querverweise oder ganz einfach vergessene Schlüssel).

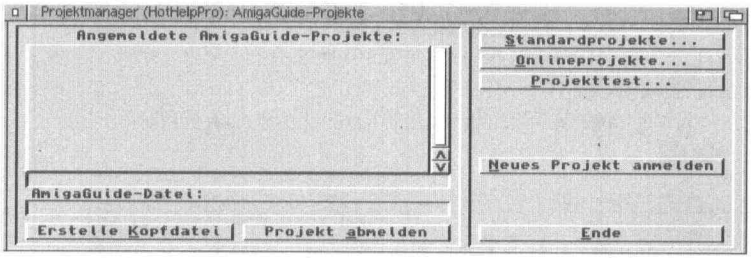


Die obere Liste enthält eine Übersicht über alle eingeschalteten Standardprojekte. Sie können hier ein oder mehrere Projekte auswählen, die dann in die untere Liste der ausgewählten Projekte übertragen werden. Wenn Sie den Test dann über das entsprechende Symbol starten, werden alle ausgewählten Projekte der Reihe nach untersucht. Dazu wird ein eigenes Fenster geöffnet, in dem Sie den Fortschritt des Testvorgangs verfolgen können. Alle gefundenen Fehler werden in einer Liste angezeigt. Außerdem werden alle Fehlermeldungen zusätzlich in eine Protokolldatei eingetragen, die Sie nach Abschluß des Tests durchlesen können.

10.4.4. AmigaGuide-Projekte

AmigaGuide ist das von Commodore favorisierte Hilfssystem, das seit OS 3.0 auch mit dem Betriebssystem zusammen ausgeliefert wird. AmigaGuide hat jedoch zwei große Probleme: zum einen ist er noch lange nicht für jeden Anwender verfügbar, da nicht jeder über OS 3.0 oder höher verfügt; zum anderen bietet er keinen besonderen Komfort - er ist nicht viel mehr als ein verbessertes Textanzeigeprogramm und kann in der aktuellen Version noch nicht einmal vernünftig über die Tastatur gesteuert werden.

Das AmigaGuide-Format ist aber dennoch interessant, da eine wachsende Anzahl von Programmen (vor allem in der Public Domain, aber auch einige kommerzielle Anwendungen) über Dokumentationen im AmigaGuide-Format verfügt. Aus diesem Grund können nun auch AmigaGuide-Dateien als normale Projektdateien in das HotHelp-System eingebunden werden. Sie können dann beim Ansehen dieser Texte den vollen Komfort ausnutzen, den HotHelp Ihnen bietet.

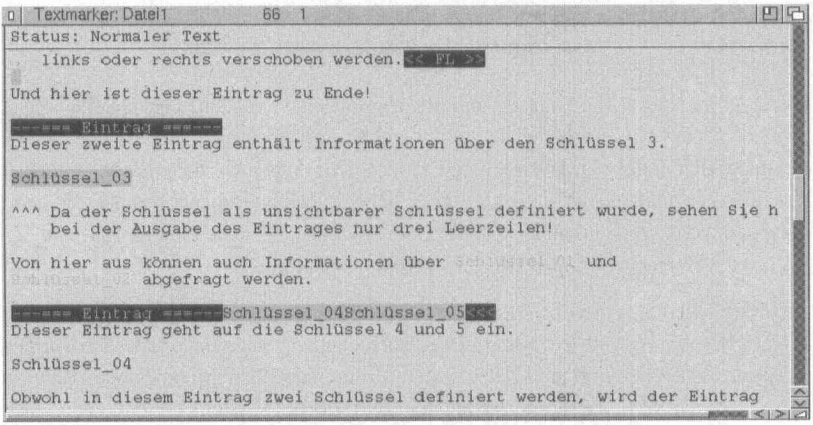


Dieses Modul dient nun dazu, eine AmigaGuide-Datei als HotHelp-Projekt anzumelden. Sie können über das Symbol 'Neues Projekt anmelden' ein Datei-Auswahlfenster öffnen, in dem Sie die gewünschte AmigaGuide-Datei auswählen können. HotHelpPro untersucht dann die Datei, erstellt eine Schlüsselübersicht und meldet die Datei unter dem in ihr angegebenen Namen als neues HotHelp-Projekt an. Von jetzt an können Sie in jedem HotHelp-Fenster auf diesen AmigaGuide-Text zugreifen, als ob es sich um ein normales HotHelp-Projekt handeln würde.

In der großen Liste sehen Sie eine Übersicht über alle bereits angemeldeten AmigaGuide-Projekte. Wählen Sie eines davon aus, wird der Name der entsprechenden AmigaGuide-Datei in dem Feld darunter angezeigt. Sie können nun mittels der beiden darunterliegenden Symbole das Projekt wieder abmelden (dadurch wird die Datei wieder aus dem HotHelp-System entfernt, aber nicht gelöscht) oder die Kopfdatei neu erstellen. Diese Aktion ist nur notwendig, wenn die AmigaGuide-Datei seit der Anmeldung noch geändert wurde, was normalerweise nicht der Fall sein sollte. HotHelp macht Sie jedoch automatisch darauf aufmerksam, wenn die Kopfdatei neu erstellt werden muß.

10.5. HotHelpMarker - der Projekte-Editor

Dies ist das umfangreichste der HotHelpTools, mit dessen Hilfe Sie ohne weiteres eigene Projekte mit Hilfstexten zu beliebigen Themen anfertigen können. Sie finden dazu im folgenden zuerst einen Abschnitt über die Grundlagen der Entwicklung von Hilfstexten, bevor dann auf die Bedienung des Markers eingegangen wird.



10.5.1. Grundlagen der Projekterstellung

Beim Aufbau von Hilfstexten gibt es zwei grundsätzlich verschiedene Möglichkeiten: Interpretieren und Kompilieren. Im ersten Fall handelt es sich bei den Hilfstexten um ganz normale ASCII-Texte, in dem bestimme Bereiche besonders markiert werden (z.B. als Schlüssel oder Querverweise). Wird ein solcher Hilfstext dargestellt, muß er nach dem Einlesen zuerst auf diese Markierungen hin untersucht werden, bevor der Text angezeigt werden kann. Diese Methode hat den Vorteil, daß die Texte schnell änderbar sind und auch ohne das spezielle Hilfsprogramm angezeigt werden können. Nachteilhaft sind jedoch die längere Wartezeit beim Einlesen und die Tatsache, daß Texte nicht komprimiert werden können, da sie ja im ASCII-Format vorliegen müssen.

Beim Kompilieren werden die Hilfstexte ebenfalls als ASCII-Texte mit besonders markierten Bereichen erstellt. Diese Texte (die als Quelldateien bezeichnet werden, da sie den Ausgangspunkt der Hilfsdateien darstellen) werden dann jedoch von einem Übersetzer in ein Binärformat umgesetzt. Die Texte können dann zwar nur noch mit dem jeweiligen Hilfsprogramm dargestellt werden; der Vorteil dieser Methode liegt jedoch in den kürzeren Ladezeiten der einzelnen Texte (die Suche nach den Markierungen wurde ja schon bei der Übersetzung durchgeführt) und der Möglichkeit, die Texte zu komprimieren, wodurch viel Platz eingespart werden kann.

Da diese Vorteile (kürzere Ladezeit und um durchschnittlich 50 % reduzierter Platzbedarf) die Nachteile weit übertreffen, wurde für HotHelp die Methode der kompilierten Hilfstexte gewählt.

Die grundlegenden Schritte bei der Erstellung eines Projektes sehen also so aus:

- ☞ Schreiben der Hilfstexte (Quelltexte)
- ☞ Markieren bestimmter Textbereiche als Schlüssel, Querverweise, ...
- ☞ Übersetzen der mit Markierungen versehenen Texte

Sie finden dazu im folgenden eine Einführung in die Erstellung von Projekten, die Ihnen Schritt für Schritt erklärt, wie Sie Hilfsdateien mit eigenen Texten erstellen können. Dabei werden nur die wichtigsten Menüs und Bedienungselemente erläutert, die für die jeweiligen Arbeitsschritte benötigt werden. Informationen über die restlichen Bedienungselemente können Sie jedoch jederzeit (wie weiter oben beschrieben) über die Help-Taste abrufen.

Während der Installation haben Sie die Möglichkeit, die Quelltexte von zwei Beispielprojekten zu installieren. Das erste Beispielprojekt beschreibt mit ausführlichen Kommentaren den Aufbau von Quelltexten und die Verwendung der verschiedenen Marken, während das zweite die sinnvolle Benutzung von Schlüsselgruppen demonstriert. Vorgabemäßig werden die Beispieldateien im Verzeichnis 'Work:HotHelpDemoProjekte' in zwei Unterverzeichnissen abgelegt. Wenn Sie die folgenden Kapitel durcharbeiten, können Sie jederzeit über den HotHelpMarker die Quelltexte der beiden Beispielprojekte darstellen, um einige praktische Beispiele für die Ausführungen dieser Anleitung zu erhalten.

10.5.2. Hintergrund

Fangen Sie am besten damit an, sich Gedanken über das Thema Ihres Projektes zu machen. Überlegen Sie sich, ob (und wenn ja, wie) Sie die Informationen in Teilbereiche aufteilen können. Falls eine solche Aufteilung sinnvoll ist, sollten Sie Ihr Projekt in Kapitel (und ggf. Unterkapitel) aufteilen. HotHelp unterstützt den Aufbau von Kapitelebenen mit beliebiger Tiefe. Machen Sie sich ggf. eine Skizze, aus der der spätere Aufbau des Projektes hervorgeht. Dies ist um so wichtiger, je komplexer der Aufbau des Projektes sein soll - wenn Sie z.B. eine Aufteilung in Kapitel, Unterkapitel etc. vornehmen wollen, sollten Sie sich schon von Anfang an darüber klar sein, wie diese Struktur aufgebaut sein soll. Spätere Änderungen der Struktur sind meist sehr aufwendig und bergen das Risiko von logischen Fehlern im Kapitelaufbau.

Verfeinern Sie nun die vorgenommene Aufteilung, bis sich einzelne, nicht weiter teilbare Gebiete herauskristallisieren. Diese Bereiche können Sie sich als die späteren Einträge des Projektes vorstellen - Texte, denen sich ein oder mehrere eindeutige Schlüsselbegriffe zuordnen lassen und die alle Informationen über den entsprechenden Begriff beinhalten.

10.5.3. Eingabe der Texte

Nachdem Sie sich über den Aufbau Ihres Projektes klar geworden sind, sollten Sie die entsprechenden Texte (die sog. Quelltexte) erstellen. Sie können dazu einen beliebigen Editor oder eine Textverarbeitung verwenden (letztere muß Texte im üblichen ASCII-Format abspeichern können).

Die Quelldateien

Ein Projekt kann sich aus beliebig vielen Dateien mit Quelltexten zusammensetzen. Sie können also alle Texte eines Projektes in einer Datei zusammenfassen oder z.B. Dateien anhand der weiter oben beschriebenen logischen Gliederung bilden, wobei sich in jeder Datei alle Texte zu dem entsprechenden Teilgebiet befinden. Für kleine Projekte ist es sicher am einfachsten, alle Texte in einer Datei zusammenzufassen, während bei großen Projekten bei dieser Methode wahrscheinlich schnell die Übersichtlichkeit verloren geht. Entscheiden Sie sich selbst für den für Ihre Zwecke besten Weg!

Die Dateinamen sind ebenfalls beliebig; allerdings müssen Sie zwei Besonderheiten beachten: Die Namen dürfen nicht länger als 26 Zeichen sein und keine Datei darf unter dem Namen 'HotHelp-Compiler.Pref' oder 'HotHelpCompiler.Titles' abgelegt werden - diese beiden Dateien werden dazu verwendet, HotHelp-interne Daten zu den Quelldateien jedes Projektes festzuhalten.

Alle Quelldateien des Projektes müssen unbedingt in einem einzigen Verzeichnis zusammengefaßt werden. In diesem Verzeichnis sollten sich auch keine Quelltexte anderer Projekte befinden; das Verzeichnis muß ausschließlich für die Quelldateien dieses einen Projektes reserviert sein! Am einfachsten ist es, wenn Sie dem Verzeichnis den Namen geben, den auch Ihr Projekt später tragen soll.

Wenn Sie eine Festplatte besitzen, sollten Sie sich ein eigenes Verzeichnis auf der Platte anlegen (z.B. 'WORK:HotHelpSource'), in dem Sie dann alle Unterverzeichnisse mit den Quelldateien Ihrer verschiedenen Projekte ablegen können. Wenn Sie dann noch die Zeile 'Assign HHC: WORK:HotHelpSource' in Ihre Startup-Sequence einsetzen, können Sie später vom HotHelp-Textmarker aus über das HHC:-Aktionssymbol des Dateiauswahl-Dialogs schnell Ihre Quelldateien erreichen.

Diskettenbenutzer sollten eine eigene Diskette für die Quelldateien der Projekte verwenden, auf der dann die einzelnen Verzeichnisse für die verschiedenen Projekte angelegt werden können. Geben Sie der Diskette am besten den Namen 'HHC', dann können Sie (wie oben beschrieben) schnell auf die Verzeichnisse zugreifen.

(Wie Sie Verzeichnisse anlegen, neue Disketten formatieren und Disketten umbenennen können, entnehmen Sie bitte Ihrem Amiga-Benutzerhandbuch).

Die Texte

Geben Sie nun die Texte ein. Sie können sich dabei ganz auf die Inhalte konzentrieren - das Markieren von Bereichen muß nun nicht mehr (wie in der vorherigen HotHelp-Version) von Hand durch Einfügen von mehr oder weniger einprägsamen Kennungen in den Text durchgeführt werden - dafür wird im nächsten Schritt der HotHelp-Textmarker herangezogen!

10.5.4. Markieren der Texte

Der HotHelp-Textmarker

Starten Sie nun bitte den HotHelp-Textmarker. Rufen Sie als erstes aus dem Projekt-Menü den Punkt 'Projektverzeichnis' auf, um das Verzeichnis festzulegen, mit dem Sie arbeiten wollen. Wählen Sie dazu im Datei-Auswahlfenster das Verzeichnis aus, in dem sich Ihre eben erstellten Quelldateien

befinden und bestätigen Sie die Auswahl über 'OK'. Da das neue Verzeichnis noch keine Voreinstellungs-Datei beinhaltet, werden Sie gefragt, ob eine solche Datei mit Vorgabeeinstellungen angelegt werden soll. Beantworten Sie die Frage mit 'Ja'. Nun können Sie über den Menüpunkt 'Öffnen' aus dem Projekt-Menü eine Ihrer Quelldateien öffnen. Sie wird im Editorfenster dargestellt.

Während der Bearbeitung Ihrer Texte öffnen sich ggf. eine Reihe von Dialogfenstern, in denen Sie bestimmte Eingaben vornehmen müssen. Drücken Sie in einem dieser Fenster die Hilfetaste, so öffnet das Programm ein HotHelp-Fenster mit einem Hilfstext über das jeweilige Fenster, in dem alle Bedienungselemente erklärt werden.

Aufteilung des Textes

HotHelp bietet die Möglichkeit, die Hilfstexte als Fließtext zu verwalten. Dabei müssen bei der Erstellung der Texte jedoch einige Regeln beachtet werden. Ist Ihnen das zu umständlich, können Sie die Fließtext-Übersetzung auch ausschalten. Wählen Sie dazu im Voreinstellungen-Untermenü des Projekt-Menüs den Punkt 'Editieren' an und schalten Sie im daraufhin erscheinenden Optionen-Fenster das Auswahlfeld 'Erzeuge Fließtext' aus. Wenn Sie diese Einstellung verwenden, behält HotHelp bei der Übersetzung die zeilenweise Aufteilung Ihrer Quelltexte bei. Die Texte werden dann genau so ausgegeben, wie Sie sie im Quelltext angelegt haben. Ist eine Zeile dann allerdings zu lang, muß der Leser den sichtbaren Textausschnitt über die Rollsymbole von Hand nach rechts verschieben, wenn er das Zeilenende lesen möchte.

Wenn Sie Ihre Hilfstexte als Fließtext übersetzen lassen, faßt HotHelp alle direkt aufeinander folgenden Zeilen zu Absätzen zusammen. Absätze werden beendet durch Leerzeilen oder durch die Absatzende-Kennung. Sie müssen nun also entweder zwischen allen Absätzen Leerzeilen einfügen oder (wenn zwei Absätze ohne Leerzeile aufeinander folgen sollen) die Absatzende-Kennung hinter dem ersten der beiden Absätze einfügen. Bewegen Sie dazu den blinkenden Cursor hinter das Ende des ersten der beiden Absätze, die nicht durch eine Leerzeile getrennt werden sollen. Wählen Sie dann aus dem Bearbeiten-Menü im Untermenü 'Kennung einfügen' den Punkt 'Absatzende' (alternativ können Sie auch die Tastenkombination <Alt>-<F4> benutzen). Vor dem Cursor erscheint der Text '<<<', der farblich hervorgehoben ist. Sie haben damit hinter dem ersten Absatz eine Absatzende-Kennung eingefügt.

In den meisten Fällen kann allerdings auf diese Kennung verzichtet werden, wenn Leerzeilen zwischen den Absätzen eingefügt werden.

Einteilung in Einträge

Als nächstes sollten Sie nun die einzelnen Einträge (das sind die einzeln darstellbaren Hilfstexte) innerhalb der Datei voneinander abgrenzen. Hierzu gehen Sie genauso vor wie beim Einfügen der Absatzende-Kennung: plazieren Sie den Cursor ans Ende des ersten Eintrags und wählen dann im Untermenü 'Kennung einfügen' den Punkt 'Eintragsende' an. Es erscheint der farblich hervorgehobene Text '—=== Eintrag ===—', der ein Eintragsende symbolisiert. Fügen Sie nun auch hinter allen weiteren Einträgen die Kennungen ein (auch hinter dem letzten Eintrag, also am Ende der Datei!). Damit haben Sie den ersten wichtigen Schritt vollzogen: der Text der Datei wurde in eine Reihe unabhängiger Einträge aufgeteilt.

Die Schlüssel

Als nächstes muß jedem Eintrag mindestens ein eindeutiger Schlüsselbegriff zugewiesen werden, unter dem der Eintrag später angesprochen werden kann. Bewegen Sie dazu den Cursor an den Anfang der Datei. Wählen Sie nun aus dem Bearbeiten-Menü im Untermenü 'Bereichsmarken' den Punkt 'Schlüssel' an. Der HotHelp-Textmarker zeigt jetzt in der Titelzeile des Fensters an, daß die Schlüssel-Marke gesetzt ist. Suchen Sie nun im ersten Eintrag nach dem Begriff, der als Schlüssel für den Eintrag verwendet werden soll. Meist taucht dieser Begriff schon in der Überschrift des Eintrags auf (falls er eine solche hat; andernfalls können Sie jetzt noch eine eingeben). Positionieren Sie den Cursor dann auf das erste Zeichen des gewünschten Schlüsselbegriffs und wählen aus dem Bearbeiten-Menü den Menüpunkt 'Bereich markieren' und daraus den Untermenüpunkt 'Blockanfang'. Bewegen Sie den Cursor dann auf das letzte Zeichen des Schlüsselbegriffs und wählen aus demselben Untermenü 'Blockende'. Der so markierte Begriff wird farblich hervorgehoben.

Gehen Sie jetzt alle Einträge durch und weisen Sie jedem Eintrag mindestens einen Schlüssel zu. Enthält ein Eintrag keinen Begriff, der als Schlüssel passend wäre, können Sie natürlich nachträglich noch einen eingeben. Ebenso steht es Ihnen natürlich auch frei, noch weitere Änderungen am Text vorzunehmen. Bei größeren Änderungen sollten Sie allerdings die Datei abspeichern und in Ihren Editor laden, da der HotHelp-Textmarker nicht als Editor konzipiert wurde und daher auch keine umfangreichen Editiermöglichkeiten bietet. Zur Korrektur von Tippfehlern oder Eingabe kleinerer Änderungen eignet er sich allerdings so gut wie ein richtiger Editor.

Eine besondere Rolle nimmt noch der Schlüssel 'Startseite' ein. Wird ein Projekt aus der Projektübersicht ausgewählt, sucht HotHelp als erstes einen Schlüssel mit dem Namen 'Startseite' in dem gewählten Projekt. Wird er gefunden, stellt das Programm den zugehörigen Eintrag dar. Existiert kein solcher Schlüssel, wird statt dessen eine Übersicht über alle Schlüssel des Projektes dargestellt. Im Normalfall wird daher ein Eintrag mit dem Schlüssel 'Startseite' dazu verwendet, einleitende Informationen über den Inhalt des Projektes zu geben, von denen aus dann über Querverweise weitere Texte erreicht werden können.

Querverweise

Sie haben nun eine Reihe von separaten Einträgen erzeugt, die alle über einen oder mehrere Schlüsselbegriffe angesprochen werden können. Momentan besteht aber noch kein Zusammenhang zwischen diesen Einträgen. Dies kann über das Einfügen von Querverweisen geändert werden; auf diese Weise bietet sich während des Lesens eines Eintrags die Möglichkeit, in einen anderen Text mit verwandten Informationen zu wechseln.

Um Texte mit Querverweisen zu versehen, gehen Sie vor wie beim Markieren der Schlüssel - Sie wählen lediglich statt 'Schlüssel' die Bereichsmarke 'Querverweis' aus. Gehen Sie dann die Einträge nacheinander durch. Wenn Sie irgendwo auf einen Begriff stoßen, den Sie anderswo als Schlüssel markiert haben, können Sie aus dem Begriff einen Querverweis machen. Wird der Verweis später bei der Anzeige des Textes ausgewählt, wird der zu dem entsprechenden Schlüssel gehörende Eintrag angezeigt. Genauso können Sie natürlich noch explizit Querverweise einfügen, in denen Sie auf Einträge mit verwandten Informationen hinweisen. Grundregel ist jedoch, daß zu jedem Querverweis auch ein Schlüssel gleichen Namens existieren muß!

Grafiken

Hilfstexte können durch das Einfügen von Grafiken wesentlich attraktiver gestaltet werden. Als erstes müssen Sie dazu die entsprechende Grafik zeichnen. Bedienen Sie sich dazu bitte eines handelsüblichen Grafikprogrammes. Speichern Sie dann die fertige Grafik ab. Am sinnvollsten ist es, wenn Sie dazu im Verzeichnis mit den Projekt-Quelltexten ein Unterverzeichnis anlegen, in dem dann alle Grafiken des jeweiligen Projektes abgespeichert werden können.

Wählen Sie dann aus dem 'Kennung einfügen'-Untermenü den Punkt 'Grafik'. Der HotHelpMarker öffnet ein Fenster, in dem Sie den Namen der Grafikdatei eingeben oder über ein Dateiauswahlfenster komfortabel auswählen können. Unter dem Eingabefeld wird eine Vorschau der ausgewählten Grafik angezeigt.

Schriftarten

Um bestimmte Teile des Textes noch besonders hervorzuheben (Überschriften, wichtige Passagen, ...), können Sie diese fett, kursiv, unterstrichen oder in jeder beliebigen Kombination dieser Attribute darstellen lassen. Dies wird über das Schriftarten-Untermenü im Bearbeiten-Menü gesteuert.

Möchten Sie z.B. alle Überschriften fett und unterstrichen ausgeben, gehen Sie dazu wie folgt vor: Sie schalten zuerst im Bereichsmarken-Untermenü die noch von vorhin gesetzte Marke aus, indem Sie sie nochmals anwählen. Die Anzeige der aktuellen Bereichsmarke in der Titelleiste des Fensters wird daraufhin gelöscht. Danach wählen Sie im Schriftarten-Menü nacheinander die Punkte 'Unterstrichen' und 'Fett' aus. In der Titelleiste erscheinen die Kürzel 'U' und 'F' für diese beiden Schriftarten. Nun setzen Sie den Cursor an den Anfang einer Überschrift und markieren Sie diese wie gehabt (die Bereichsmarke mußte vorher ausgeschaltet werden, da ansonsten die Überschrift nicht nur fett und unterstrichen, sondern auch noch mit dieser Bereichsmarke markiert worden wäre...).

Sie können auf diese Weise jedem Textbereich beliebige Textattribute zuordnen. Um ein solches Attribut wieder zu löschen, wählen Sie im Schriftarten-Menü 'Normaler Text' aus und markieren den entsprechenden Bereich. Wollen Sie die Textattribute beim Markieren unverändert lassen, schalten Sie einfach alle Optionen im Schriftarten-Menü wieder aus (die Anzeige in der Titelleiste ist dann leer).

Die Übersetzung

Damit haben Sie die grundlegenden Arbeitsschritte kennengelernt, um ein einfaches Projekt zu erstellen. Wählen Sie nun aus dem Projekt-Menü 'Speichern' aus, so sichert der HotHelp-Textmarker die Datei einschließlich aller Änderungen auf Diskette bzw. Festplatte. Falls Sie für Ihr Projekt mehrere Quelldateien erstellt haben, sollten Sie nun die übrigen Dateien in ähnlicher Weise behandeln. Sind alle Dateien mit Marken versehen und wieder abgespeichert, kann die Übersetzung beginnen.

Dieser Schritt ist recht einfach: wählen Sie aus dem Übersetzen-Menü im Untermenü 'Dateien übersetzen' den Punkt 'Alle ...'. Der HotHelp-Textmarker lädt nun den HotHelp-Übersetzer nach, der die eigentliche Übersetzung durchführt. Ist die Übersetzung abgeschlossen, können Sie das Fenster des Übersetzers wieder schließen. Falls Fehler aufgetreten sind, öffnet sich automatisch das Überset-

zungsfehler-Fenster, in dem alle Fehler aufgelistet sind. Sie können von hier aus die fehlerhaften Zeilen einfach editieren.

Haben Sie alle Fehler beseitigt, wählen Sie am besten den Menüpunkt 'Geänderte ...' aus dem 'Dateien übersetzen'-Untermenü. Im Gegensatz zu 'Alle ...' werden hierbei nur die Dateien übersetzt, die seit dem letzten Übersetzungsvorgang noch geändert worden sind. Diese Option ist recht praktisch, wenn Sie mit vielen Quelldateien arbeiten, da erfahrungsgemäß immer nur in einigen Dateien Änderungen vorgenommen werden. Der Übersetzer verläßt sich allerdings darauf, daß die Systemzeit immer korrekt gestellt ist - ist das bei Ihnen nicht der Fall, kann es zu unerwünschten Effekten kommen. Sie sollten dann besser auf diese Option verzichten!

10.5.5. Resümee

Natürlich läßt sich die Erstellung eines Projektes nicht immer genau so in Schritte einteilen, wie dies hier beschrieben wurde. Schritt 2 (Schreiben der Texte) und Schritt 3 (Markieren und Übersetzen) werden wahrscheinlich häufig wiederholt nacheinander ausgeführt, da es erfahrungsgemäß nicht möglich ist, alle gewünschten Texte auf einmal einzugeben. Oftmals stellt sich erst später heraus, daß einige Gebiete nur ungenügend behandelt oder ganz ausgelassen wurden, obwohl sie für das Thema des Projektes noch von Interesse sind. Beim Verteilen der Querverweise ergibt sich oft der Bedarf nach neuen Einträgen mit ergänzenden Informationen, die dann durch diese Querverweise angesprochen werden sollen.

Wenn sie einen Text wieder in den Editor laden, den Sie vorher im HotHelp-Textmarker mit Marken versehen haben, werden Sie feststellen, daß der Text geändert wurde - der HotHelp-Textmarker hat eine Reihe von Kennungen eingefügt, mit denen die Markierung der im Editorfenster sichtbaren Bereiche in der Datei festgehalten wird. Tatsächlich handelt es sich hierbei um die Marken, die unter Version 2.00 von HotHelp noch von Hand in den Text eingefügt werden mußten! Wenn Sie einen mit solchen Marken versehenen Text von einem Editor aus ändern, sollten Sie darauf achten, die Kennungen selber nicht zu verändern - andernfalls kann es beim nächsten Mal, wenn die Datei vom HotHelp-Textmarker geladen wird, zu Problemen kommen!

Wenn Sie die Arbeit an Ihrem Projekt abgeschlossen haben, sollten sie über den Menüpunkt 'Projekt überprüfen' aus dem Übersetzen-Menü einen Test Ihres Projektes durchführen. Dabei werden Sie z.B. darauf hingewiesen, wenn ein Begriff mehrfach als Schlüssel definiert wurde, wenn zu einem Querverweis kein passender Schlüssel existiert oder wenn gar mehrere passende Schlüssel existieren. Dies alles kann auf logische Fehler (oder auch einfach Schreibfehler) in Ihrem Projekt hindeuten.

10.5.6. Weitere Gestaltungsmöglichkeiten

Hier werden einige weitergehende Möglichkeiten erläutert, um Hilfstexte zu gliedern und in eine bestimmte Struktur zu bringen.

Titel

An jeden Eintrag kann ein Titel vergeben werden, der bei der Anzeige des Textes in der Titelzeile des HotHelp-Fensters hinter dem Projekt- und dem Schlüsselbegriff angezeigt wird.

Kapitel, Vorgänger und Nachfolger

Diese drei Bereiche wirken sich bei der Darstellung auf das Kapitel- und die beiden Pfeil-Aktionssymbole des HotHelp-Fensters aus. Sie werden im Prinzip genauso verwendet wie Querverweise, allerdings sind die so markierten Bereiche nicht mehr im übersetzten Text sichtbar. In jedem Eintrag dürfen diese drei Bereiche jeweils höchstens einmal auftreten.

Während normale Querverweise im Hilfstext sichtbar sind und über die Maus oder die Tastatur ausgewählt werden, benutzt HotHelp die über diese drei Marken definierten Querverweise, wenn das Kapitel- oder eines der beiden Pfeil-Aktionssymbole vom Anwender betätigt wird. Sie können also auf diese Weise jedem Text eine Kapitelübersicht, einen Vorgänger und einen Nachfolger zuordnen.

Projekt- und ARexx-Querverweise

Über Projekt-Querverweise kann das Problem mehrerer gleichlautender Schlüssel in verschiedenen Projekten gelöst werden. Möchten Sie z.B. einen Querverweis auf das Wort 'Task' einfügen, ergibt sich das Problem, daß dieser Schlüssel sowohl im Projekt 'Glossar' als auch in 'Libs 3.0' auftaucht - natürlich mit gänzlich unterschiedlichen Bedeutungen. In diesem Fall können Sie über den Projekt-Querverweis auch noch den Namen des Projektes angeben, auf das sich der Querverweis beziehen soll.

ARexx-Querverweise ermöglichen es, ARexx-Kommandos in einen HotHelp-Text einzubinden. Wählt der Anwender dann den Querverweis aus, wird der Befehl an ARexx übermittelt und ausgeführt. Wahlweise kann auch ein ARexx-Host angegeben werden, an den das ARexx-Kommando dann geschickt wird. Voraussetzung dafür ist allerdings, daß ARexx auf Ihrem System installiert ist!

Schlüsselgruppen

Schlüsselgruppen stellen eine Möglichkeit dar, die Entwicklung umfangreicher Projekte mit mehreren Gliederungsstufen (Kapitel, diesen zugeordnete Unterkapitel, etc.) zu erleichtern. Jedes Projekt kann über bis zu 255 unterschiedliche Schlüsselgruppen verfügen; jeder Schlüssel kann einer oder mehreren dieser Gruppen zugeteilt werden.

Solche Gruppen können nun auf dreierlei Weisen verwendet werden. Über die Gruppenlisten-Kennung kann eine Übersicht aller Schlüssel einer oder mehrerer Gruppen in den Text eingefügt werden. Jeder der Schlüssel wird als Querverweis angezeigt, so daß der zugehörige Text vom Benutzer auf die übliche Weise ausgewählt werden kann.

Weiterhin kann jeder Gruppe ein Titel (ein sog. Gruppentitel) zugeordnet werden. Wird ein Hilfstext zu einem Schlüssel angezeigt, der sich in einer mit einem Titel versehenen Schlüsselgruppe befindet, wird dieser Gruppentitel in der Titelleiste des Fensters zwischen dem Projekt- und dem Schlüssel-Namen angezeigt. Betätigt der Benutzer das Kapitel-Aktionssymbol im HotHelp-Fenster, wird der Gruppentitel als Schlüsselbegriff verwendet und der entsprechende Hilfstext angezeigt, sofern ein Schlüssel dieses Namens in dem Projekt definiert wurde. Andernfalls wird die Startseite

des Projektes angezeigt. Die Wirkung dieses Gruppentitels kann durch explizite Angabe einer Kapitel-Marke (s.o.) überschrieben werden: eine mit der Kapitel-Marke definierte Verknüpfung hat Vorrang vor der durch einen Gruppentitel hergestellte Beziehung.

Gruppentitel können über den entsprechenden Menüpunkt im Projekt-Menü editiert werden. Bitte beachten Sie, daß zwischen den normalen Titeln und den Gruppentiteln keinerlei Zusammenhang besteht! Normale Titel werden lediglich in der Titelleiste des Fensters angezeigt, haben jedoch keinerlei weitere Wirkung!

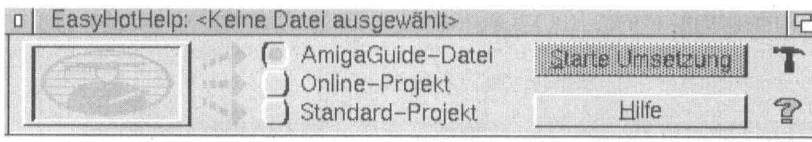
Gruppen können außerdem dazu dienen, das Problem mehrerer gleichlautender Schlüssel zu lösen. Normalerweise sollte ein Projekt zwar nicht mehrere Schlüssel mit gleichem Namen, aber unterschiedlichen Texten beinhalten. Ist dies aber doch einmal der Fall (wie es sich z.B. bei dem Projekt Libs & Devs nicht umgehen ließ, wo es Überschneidungen zwischen Struktur-, Funktions- und Konstantennamen gibt), stellt sich die Frage: Wie kann ein Querverweis sich eindeutig auf einen der gleichlautenden Schlüssel beziehen?

Die Antwort: Jeder der Querverweis-Marken kann ebenfalls eine Gruppennummer zugeordnet werden. Geben Sie nun demjenigen der gleichlautenden Schlüssel, den Sie durch einen Querverweis ansprechen möchten, eine eindeutige Nummer, können Sie sich im entsprechenden Querverweis über diese Nummer direkt auf den gewünschten Schlüssel beziehen.

Schlüsselgruppen sind keine sehr einfache Materie und da ein Beispiel mehr sagt als tausend Worte, soll hier nur noch auf das zweite der beiden Beispielprojekte verwiesen werden. Es zeigt an dem leicht nachvollziehbaren Thema von Kochrezepten mit vielen erklärenden Texten, wie Gruppen sinnvoll in HotHelp-Projekten angewendet werden können. Am besten starten Sie jetzt den HotHelpMarker, machen das Verzeichnis 'BeispielProjekt2' zum Projektverzeichnis und arbeiten die Dateien in diesem Verzeichnis der Reihe nach durch (beginnen Sie mit der Startseite und fahren dann mit der italienischen Küche fort).

10.6. EasyHotHelp

EasyHotHelp ist ein Werkzeug, mit dessen Hilfe Sie sehr einfach schon bestehende Dokumente als HotHelp-Projekte anmelden können. Sie können damit alle Textdateien, die im ASCII-Format vorliegen (speziell auch AmigaGuide-Dateien), sehr einfach umsetzen - z.B. die Anleitungen von Public-Domain-Programmen. Voraussetzung für die Verwendung dieses Programms ist allerdings, daß Sie mindestens Betriebssystemversion 2.0 verwenden.



Nach dem Start öffnet das Programm auf der Workbench ein kleines Fenster. Ergreifen Sie nun das Piktogramm des Textes, den Sie bearbeiten möchten, und legen es in das EasyHotHelp-Fenster. Das Piktogramm wird in dem Fenster dargestellt, und der Name der Datei erscheint in der Titelleiste. Sie können diese Datei nun auf drei verschiedenen Wegen umsetzen, indem Sie den entsprechenden Druckknopf anwählen und dann die Umsetzung über das entsprechende Symbol starten:

- Als AmigaGuide-Datei. Dies empfiehlt sich natürlich nur, wenn es sich auch wirklich um eine Datei im AmigaGuide-Format handelt - ansonsten zeigt der Übersetzer Ihnen eine entsprechende Fehlermeldung an. AmigaGuide-Dateien haben üblicherweise die Endung '.guide'.
- Als Online-Projekt. Die Textdatei wird als Eintrag eines Online-Projektes verwendet (siehe auch Kapitel 7). Das Programm verhält sich dabei genauso, als ob Sie die Datei in einen der unterstützten Editoren geladen, als Block markiert und dann über <Ctrl>-<Amiga>-<s> übernommen hätten. Sie haben somit also die Möglichkeit, Online-Projekte zu definieren, auch wenn Sie über keinen der genannten Editoren verfügen.
- Als Standardprojekt. Wie schon in Kapitel 10.5 erläutert wurde, müssen - um aus einer normalen Textdatei ein Standardprojekt zu erzeugen - bestimmte Bereiche des Textes als Schlüssel, Querverweise, etc. markiert werden.

Um solche Markierungen in den Text einzusetzen, kann normalerweise der HotHelpMarker benutzt werden. Dieser unterstützt zwar alle Möglichkeiten der Projekterzeugung, kann durch seine Vielfalt an Funktionen für den Einsteiger jedoch auch sehr verwirrend sein. Sie finden daher hier eine Art 'kalorienarmer HotHelpMarker light'.

EasyHotHelp stellt den geladenen Text dazu in einem weiteren Fenster dar, wo Sie durch einfaches Anklicken und Ziehen mit der Maus Schlüssel, Querverweise und Eintragsende setzen und den fertig bearbeiteten Text dann sofort übersetzen können. Im Falle von Übersetzungsfehlern können Sie diese anzeigen lassen und korrigieren.

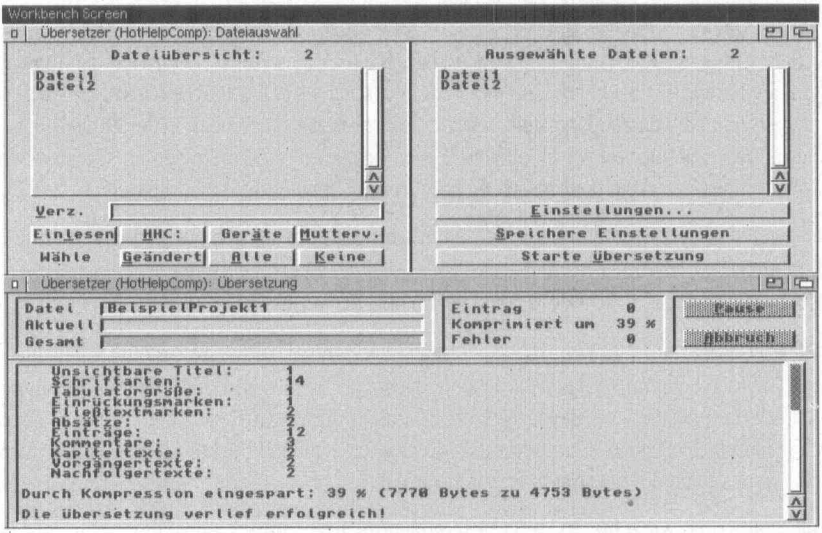
Sie können den so markierten Text auch später noch mit dem HotHelpMarker weiterbearbeiten - EasyHotHelp erzeugt ein temporäres Verzeichnis (üblicherweise in der RAM-Disk, Sie können

jedoch auch ein anderes Verzeichnis wählen), in dem der Text mit seinen Markierungen im üblichen Format abgelegt wird.

Weitere Informationen über EasyHotHelp können Sie wie üblich nach dem Start des Programms über die Hilfetaste bzw. die Hilfe-Symbole und -Menüs abfragen.

10.7. HotHelpComp - der HotHelp-Übersetzer

Dieses Programm entspricht dem 'Projekte übersetzen'-Modul des alten HotHelpManagers der Version 2.00 von HotHelp. Es wird bei der Erstellung eigener Projekt verwendet, um die mit Textmarken versehenen Quelltexte in HotHelp-Projekte zu übersetzen.



Das Programm ist ebenfalls in mehrere Fenster gegliedert, die zum einen dem eigentlichen Übersetzungsvorgang dienen, zum anderen jedoch auch weitreichende Möglichkeiten der freien Gestaltung der Markierungen bieten, die in den Quelltexten besondere Bereiche (Querverweise, Schlüssel, Schriftarten, Absatzenden etc.) kennzeichnen.

Da HotHelpComp üblicherweise im Rahmen der Übersetzung vom HotHelpMarker aufgerufen und kontrolliert wird, soll an dieser Stelle nicht weiter auf die Bedienung dieses Programms eingegangen werden.

Der "Power-User", dem das Herumwirbeln mit sinistren HotHelp-Steuercodes in Dutzenden von Quelldateien in MegaByte-großen Projekten nicht über den Kopf wächst, und der seine HotHelp-Quelldateien lieber auf herkömmliche Weise mit dem Editor erzeugt und ändert, hat jedoch die Möglichkeit, den HotHelp-Übersetzer direkt aufzurufen, ohne den "Umweg" über den HotHelp-Textmarker gehen zu müssen (Ähnlichkeiten mit dem Autor des Libs&Devs-Projektes sind weder zufällig noch unbeabsichtigt).

11. Letzte Worte

Viele Dinge wurden in diesem Handbuch nur oberflächlich angerissen - und das aus gutem Grund: HotHelp soll Ihnen schließlich das endlose Blättern in Handbüchern ersparen - warum sollten wir Sie also mit einem dicken HotHelp-Handbuch belasten? Die Projekte 'HotHelp' und 'HotHelpTools' enthalten alle Informationen, die für die Arbeit mit HotHelp und seinen Zusatzprogrammen notwendig sind. Auf diese Weise steht Ihnen auf Tastendruck hin genau die Information zur Verfügung, die Sie gerade brauchen. Zögern Sie also nicht, diese Informationsmöglichkeit zu nutzen, ehe Sie der Funktion eines rätselhaften Bedienungselementes durch Probieren auf den Grund zu gehen versuchen!

Das HotHelp-System selbst wurde unter ständiger Nutzung von HotHelp programmiert (insbesondere das Projekt Libs & Devs wurde häufig zu Rate gezogen). Es ist also praxiserprobt, und wir selbst möchten bei der Arbeit nicht mehr darauf verzichten. Wir hoffen daher, daß HotHelp auch Ihnen Ihre Arbeit am Amiga erleichtert und Ihnen - gemäß dem Motto von HotHelp - wirklich schnelle Hilfe zu jeder Zeit bietet.

In diesem Sinne wünschen wir Ihnen viel Erfolg.

Die Autoren

AMIGA

MaxonC++

68000/10/20/30/881/882/851 Assembler

MAXON
computer

1. Einleitung

MaxonASM-CLI ist eine spezielle Version des integrierten Komplettpakets MaxonASM. Es handelt sich dabei nicht um eine abgespeckte Fassung, so daß alle Features vorhanden sind. Der Assembler unterstützt alle Codes der Prozessoren 68000/10/20/30, der FPUs 68881/2 sowie der PMMUs 68851/68030. Desweiteren können Makros und Includes genutzt werden. Die Assembliergeschwindigkeit ist beim MaxonASM sehr hoch, je nach Programm erreicht er schon auf einem 68000-Amiga Geschwindigkeiten von 60000-90000 Zeilen pro Minute.

Sie erhalten mit der CLI-Version des MaxonASM also einen Assembler, den Sie neben der Verwendung mit MaxonC^{++} auch für größere Assembler-Projekte komfortabel einsetzen können.

2. Grundlagen der Assemblerprogrammierung

Eine komplette Einführung in die Assembler-Programmierung würde den Rahmen dieses Handbuches bei weitem sprengen. Bitte verstehen Sie deshalb dieses Kapitel nur als Überblick über die Möglichkeiten, die Ihnen Assembler bietet.

2.1 Warum Assembler?

Schon seit geraumer Zeit gewinnen Hochsprachen immer mehr an Bedeutung. Ein Beispiel für den Trend hin zu Hochsprachen ist das Betriebssystem des Amiga. Es wurde zum größten Teil in C, DOS sogar in BCPL verfaßt (ab OS 2.0 ist DOS in C geschrieben). Der Vorteil von Hochsprachen liegt in der einfacheren Programmierung und der geringeren Fehleranfälligkeit. Im allgemeinen sind auch Hochsprachenprogramme leichter zu lesen und zu durchschauen. Aber auch die Nachteile von Hochsprachen sollen nicht verschwiegen werden. Die Programme müssen erst von einem Compiler in eine für den Prozessor verständliche Form (Maschinensprache) übersetzt werden. Der enstehende Code ist nicht besonders schnell und meist länger als der direkt in Assembler geschriebener Programme. Assembler bietet zudem noch den Vorteil, sich nicht mit vielen verschiedenen Variablentypen auseinandersetzen zu müssen.

2.2 Der Aufbau eines Quelltextes

Die Zeile eines Quelltextes gliedert sich grob in 4 Teile:

Labelfeld	Opcodefeld	Operandenfeld	Kommentar
`Start`	`LEA`	`Label,A1`	`;Beispiel`

Ganz links, direkt in der ersten Spalte, stehen Label. Label sind Sprungmarken. Diese Sprungmarken haben die Aufgabe, die von Basic her bekannten Zeilennummern zu ersetzen und das Programm lageunabhängig assemblieren zu können. Sie dürfen für Label beliebig lange Namen verwenden. Durch mindestens ein Leerzeichen vom Rand bzw. Label getrennt beginnt das Opcode-Feld. Hier stehen die Befehle (Mnemonics) bzw. Direktiven für den Assembler. Auf den Befehl folgen die Operanden. Mehrere Operanden werden durch Kommata getrennt. Zum Schluß der Zeile können Sie noch einen Kommentar anfügen. Kommentare werden mit einem Semikolon bzw. mit einem „*“ einge- leitet. Sie können natürlich einzelne Teile einer Zeile weglassen, z.B. ist auch

```
Start       ; Zeile ohne Opcode
```

oder

```
moveq   #0,d0    ; Zeile ohne Label
```

oder

```
; Zeile ohne Label und Opcode
```

richtig.

Sie sollten Ihren Quelltext mit möglichst vielen Kommentaren ausstatten, damit Sie sich auch nach längerer Zeit noch in dem entsprechenden Programm zurechtfinden. Sehr hilfreich ist es, den Quelltext mit einem sogenannten Kommentarkopf zu versehen, in dem der Name des Programms, eine grobe Beschreibung der Funktion, die Versionsnummer und eventuell auch Copyright-Vermerke stehen.

3. Der Assembler

Er unterstützt die Programmierung der Prozessoren 68000,68010,68020 und 68030, der FPUs 68881/68882 und der MMU 68851. Selbstverständlich sind auch die Befehle zur Programmierung der im MC 68030 integrierten MMU implementiert.

Den externen Assembler rufen Sie wie folgt auf:

```
asm  [FROM] [PROG] <Quelldatei> [-o <Objektdatei>]
     [-l <Listingdatei>] [-v <Fehlerdatei>]
     [-t <Symboltabelle>] [-c <Optionen>]
```

folgende Optionen sind dabei möglich :

- **c** - es wird zwischen Groß- und Kleinschreibung unterschieden
- **w** - Warnungen werden unterdrückt
- **o** - aktiviert den Multipass-Optimierer
- **s** - es werden Symbolhunks erzeugt
- **a** - es wird absoluter Code erzeugt, die Datei ist nicht linkbar bzw. vom CLI aus ladbar !
- **e** - es gelangen nur exportierte Symbole in den Symbolhunk
- **m** - Macros werden in Listing expandiert
- **l** - es wird eine linkbare Datei erzeugt

3.1 Die Adressierungsarten des MC 68000

Der MC 68000 ist ein 16/32-Bit-Prozessor. 16 Bit deshalb, weil sein Datenbus nur 16 Bit breit ist, und 32 Bit, weil die Operationen innerhalb des Prozessors bis auf wenige Ausnahmen mit einer Breite von 32 Bit ablaufen.

Der MC 68000 verfügt über 8 Daten- und 8 Adressregister, von denen allerdings nur die ersten 7 frei zur Verfügung stehen, da das 8. Adressregister als Stackpointer benutzt wird. Die Datenregister erhalten die Bezeichnungen D0 bis D7 und die Adressregister A0 bis A7. Sie können für A7 auch die Bezeichnung SP verwenden. Weiterhin gibt es ein sogenanntes Statusregister, in dem die für bedingte Verzweigungen wichtigen Flags und im oberen Byte wichtige Bits, die den Zustand des Prozessors beschreiben, liegen. Das gesamte Register (16 Bit) erreichen Sie mit SR, das unter Byte, das die sogenannten Conditioncodes enthält, heißt CCR. Für die zwei Betriebsmodi des MC 68000, den normalen Usermode und den Supervisormode, existieren noch zwei Stackpointer zur Verwal-

tung getrennter Stackbereiche. Zum einen ist das der USP für den Usermode und zum anderen der SSP für den Supervisiormodus. Der Zugriff auf diese beiden Register sowie auf das SR (schreibend) ist privilegiert, d.h. nur im Supervisormodus erlaubt.

Adressierungsarten

Adressierungsart	Syntax alt	Syntax neu	000	010	020	030
Datenregister direkt	Dn	Dn	*	*	*	*
Adreßregister direkt (AR)	An	An	*	*	*	*
AR indirekt (ARI)	(An)	(An)	*	*	*	*
ARI mit Postinkrement	(An)+	(An)+	*	*	*	*
ARI mit Predekrement	-(An)	-(An)	*	*	*	*
ARI mit 16 Bit Distanz	d16(An)	(d16,An)	*	*	*	*
ARI mit 8 Bit Dist.,Index	d8(An,Rn.x)	(d8,An,Rn.x)	*	*	*	*
ARI,d8,Index,Scalar	d8(An,Rn.x*sc)	(d8,An,Rn.x*sc)			*	*
ARI,BD,Index	bd(An)	(bd,An)			*	*
	(Rn.x)	(Rn.x)			*	*
	bd(Rn.x)	(bd,Rn.x)			*	*
	bd(An,Rn.x)	(bd,An,Rn.x)			*	*
ARI,BD,Index,Scalar	(Rn.x*sc)	(Rn.x*sc)			*	*
	bd(Rn.x*sc)	(bd,Rn.x*sc)			*	*
	bd(An,Rn.x*sc)	(bd,An,Rn.x*sc)			*	*
Speicher indirekt Postindex	([bd])	([bd])			*	*
	([An])	([An])			*	*
	([bd,An])	([bd,An])			*	*
	([bd],Rn.x)	([bd],Rn.x)			*	*
	([bd],Rn.x*sc)	([bd],Rn.x*sc)			*	*
	([An],Rn.x)	([An],Rn.x)			*	*
	([An],Rn.x*sc)	([An],Rn.x*sc)			*	*
	([bd,An],Rn.x)	([bd,An],Rn.x)			*	*
	([bd,An],Rn.x*sc)	([bd,An],Rn.x*sc)			*	*
	([bd],od)	([bd],od			*	*
	([An],od)	([An],od)			*	*
	([bd,An],od)	([bd,An],od)			*	*
	([bd],Rn.x,od)	([bd],Rn.x,od)			*	*
	([bd],Rn.x*sc,od)	([bd],Rn.x*sc,od)			*	*
	([An],Rn.x,od)	([An],Rn.x,od)			*	*
	([An],Rn.x*sc,od)	([An],Rn.x*sc,od)			*	*
	([bd,An],Rn.x,od)	([bd,An],Rn.x,od)			*	*
	([bd,An],Rn.x*sc,od)	([bd,An],Rn.x*sc,od)			*	*
Speicher indirekt Preindex	([bd,Rn.x])	([bd,Rn.x])			*	*
	([bd,Rn.x*sc])	([bd,Rn.x*sc])			*	*
	([An,Rn.x])	([An,Rn.x])			*	*
	([An,Rn.x*sc])	([An,Rn.x*sc])			*	*
	([bd,An,Rn.x])	([bd,An,Rn.x])			*	*
	([bd,An,Rn.x*sc])	([bd,An,Rn.x*sc])			*	*
	([bd,Rn.x],od)	([bd,Rn.x],od)			*	*
	([bd,Rn.x*sc],od)	([bd,Rn.x*sc],od)			*	*
	([An,Rn.x],od)	([An,Rn.x],od)			*	*
	([An,Rn.x*sc],od)	([An,Rn.x*sc],od)			*	*
	([bd,An,Rn.x],od)	([bd,An,Rn.x],od)			*	*
	([bd,An,Rn.x*sc],od)	([bd,An,Rn.x*sc],od)			*	*
Absolut kurz	$xxx.w	($xxx).w	*	*	*	*
Absolut lang	$xxx.l	($xxx).l	*	*	*	*
PC-rel. mit 16 Bit Distanz	d16(PC)	(d16,PC)	*	*	*	*
PC-rel. mit 8 Bit Dist.,Index	d8(PC,Rn.x)	(d8,PC,Rn.x)	*	*	*	*
PC-rel. d8,Index,Scalar	d8(PC,Rn.x*sc)	(d8,PC,Rn.x*sc)			*	*

```
PC-rel., BD,Index                 bd(PC)            (bd,PC)                  *   *
                                bd(PC,Rn.x)       (bd,PC,Rn.x)               *   *
PC-rel., BD,Index,Scalar       bd(PC,Rn.x*sc)    (bd,PC,Rn.x*sc)             *   *
PC-rel.,Speicher ind. Postinx.  ([PC])              ([PC])                   *   *
                                 ([bd,PC])          ([bd,PC])                *   *
                                ([PC],Rn.x)        ([PC],Rn.x)               *   *
                               ([PC],Rn.x*sc)     ([PC],Rn.x*sc)             *   *
                               ([bd,PC],Rn.x)     ([bd,PC],Rn.x)             *   *
                              ([bd,PC],Rn.x*sc)  ([bd,PC],Rn.x*sc)           *   *
                               ([bd,PC],od)       ([bd,PC],od)              *   *
                              ([PC],Rn.x,od)     ([PC],Rn.x,od)             *   *
                             ([PC],Rn.x*sc,od)  ([PC],Rn.x*sc,od)           *   *
                             ([bd,PC],Rn.x,od)  ([bd,PC],Rn.x,od)           *   *
                            ([bd,PC],Rn.x*sc,od) ([bd,PC],Rn.x*sc,od)       *   *
PC-rel.,Speicher ind. Preinx. ([PC,Rn.x])        ([PC,Rn.x])                *   *
                               ([PC,Rn.x*sc])     ([PC,Rn.x*sc])             *   *
                               ([bd,PC,Rn.x])     ([bd,PC,Rn.x])             *   *
                              ([bd,PC,Rn.x*sc])  ([bd,PC,Rn.x*sc])           *   *
                              ([PC,Rn.x],od)     ([PC,Rn.x],od)             *   *
                             ([PC,Rn.x*sc],od)  ([PC,Rn.x*sc],od)           *   *
                             ([bd,PC,Rn.x],od)  ([bd,PC,Rn.x],od)           *   *
                            ([bd,PC,Rn.x*sc],od) ([bd,PC,Rn.x*sc],od)       *   *
Konstante unmittelbar            #k                  #k        *   *         *   *
```

Abkürzungen :

`Dn`	Dateneregister `D0...D7`
`An`	Adreßregister `A0...A7 (SP)`
`Rn.x`	Register `D0...D7` oder `A0...A7 (SP)` und optionale Länge `.W` oder `.L`
`sc`	Skalar, mit dem Indexregister multipliziert wird. Erlaubte Werte : 1,2,4 oder 8
`d16`	16-Bit-Distanz
`d8`	8-Bit-Distanz
`bd`	Base Displacement 16 oder 32 Bit
`od`	Outer Displacement 16 oder 32 Bit

Der beim MC 68000 nicht privilegierte Befehl `MOVE SR,<ea>` darf ab dem MC 68010 nur noch im Supervisor-Mode benutzt werden !

3.2 Symbole und mathematische Ausdrücke

Bei den Symbolen ist grundsätzlich zwischen globalen und lokalen zu unterscheiden. Globale Symbole gelten im gesamten Quelltext während lokale Symbole nur zwischen zwei globalen Symbolen bzw. innerhalb eines Macros gelten.

3.2.1 Globale Symbole

Globale Symbole beginnen immer miteinem Buchstaben bzw. einem Underscore (`"_"`). Danach dürfen weitere Buchstaben, Ziffern bzw. Underscores folgen. Alle Zeichen sind signifikant, d.h. die

Namen dürfen beliebig lang sein und alle Zeichen werden verglichen. Es wird grundsätzlich nicht zwischen Groß- und Kleinschreibung unterschieden.

```
DosBase dc.l    0    ;DosBase ist ein globales Label
LN_SUCC RS.L    1    ;LN_SUCC ist eine globale Variable
```

3.2.2 Lokale Symbole

Lokale Symbole gelten solange, bis wieder ein globales Symbol definiert wird. Zusätzlich gibt es einen Sonderfall, es gibt auch lokale Symbole, deren Gültigkeit auf ein Macro beschränkt ist.
Symbole gelten als lokal, wenn sie:

1. mit einer Ziffer beginnen und mit dem "$"-Zeichen enden:

```
0$  clr.b (a0)+
  dbra d0,0$       ;0$ ist ein lokales Label
```

2. mit einem Punkt beginnen:

```
.Loop    clr.b (a0)+
         dbra d0,.Loop
```

3. mit einem Backslash ("\") beginnen:

```
\Loop    clr.b (a0)+
         dbra d0,\Loop
```

Diese Möglichkeit sollte allerdings innerhalb von Macros vermieden werden, da sie zu Kollisionen mit den Parameterbezeichnungen führt.

3.2.3 Label

Als Label bezeichnet man Sprungmarken, die den Stand des PC an der Stelle, an der sie definiert wurden, repräsentieren. Sie dienen als Ziel für Verzweigungen bzw. absoluter oder PC-relativer Adressierungen. Ein Label wird definiert, wenn es in einer Zeile ganz links beginnt und keine Direktive wie **EQU**, **SET** bzw. **RS** folgt. Label dürfen sowohl lokal als auch global sein.

Beispiele:

```
Das_ist_ein_Label
doslib.name      ;auch der Punkt darf vorkommen
doslib.l         ;ist verboten, da .l als Größenangabe gilt!
.Start           ;ein lokales Label
```

3.2.4 Variablen

Variablen bekommen im Gegensatz zu den Labeln mittels einer Direktive einen Wert zugewiesen, der völlig unabhängig vom aktuellen PC ist. Variablen sind nur dann redefinierbar, d.h. man darf ihnen an beliebiger Stelle einen neuen Wert zuweisen, wenn sie mit der Direktive **SET** definiert wurden.

3.2.5 Reservierte Symbole

Es existieren vier reservierte Symbole, deren Namen Sie also nicht für eigene Symbole verwenden dürfen.

3.2.5.1 NARG

Dieses Symbol darf nur innerhalb von Macros verwendet werden und enthält immer die Anzahl der Parameter, die dem Macro übergeben wurden.

z.B.:

```
IFEQ NARG-2
...        ;diese Zeilen werden nur assembliert, wenn dem
...        ;Macro genau zwei Parameter übergeben wurden
ENDIF
```

Ein weiteres Beispiel für die Benutzung von NARG finden Sie im Demoprogramm #1.

3.2.5.2 __RS

Das Symbol **__RS** enthält den aktuellen Stand des RS-Zählers. Durch die Direktiven **RS**, **RSSET** und **RSRESET** wird dieser Zähler verändert.

z.B.:

```
   RSRESET          ; __RS = 0 !
X1 RS.L 1           ; __RS = 4
X2 RS.L 1           ; __RS = 8
Y1 RS.L 1           ; __RS = 12
Y2 RS.L 1           ; __RS = 16
   RSSET 500        ; __RS = 500
Z  EQU __RS         ; Z = 500
```

3.2.5.3 _MOVEMBYTES

Dieses Symbol gibt die Anzahl der mit dem letzten MOVEM-Befehl bewegten Bytes wieder. Dieses Symbol kann man z.B. dazu benutzen, den Stackpointer (**SP**) nach einem **MOVEM**-Befehl zu korrieren ohne die Register wieder vom Stack holen zu müssen:

```
    movem.l d0-d4/d6/a0-a2/a4,-(sp)
    ...
    ...
    beq.s   0$
    add.w   #_MOVEMBYTES,SP      ;Stackpointer korrigieren
    rts
0$  movem.l (sp)+,d0-d4/d6/a0-a2/a4
    rts
```

3.2.6 Mathematische Ausdrücke

Der Assembler berechnet mathematische Ausdrücke unter Berücksichtigung der Priorität der mathematischen Operationen. Sie dürfen auch Klammern benutzen, die maximal 32mal ineinander geschachtelt sein dürfen.

3.2.7 Zahlenformate

Alle Zahlen sind grundsätzlich vom Format Integer 32 Bit(-$80000000 <= n <= $7FFFFFFF bzw. 0 <= n <= $FFFFFFFF)

1. Dezimalzahlen

```
123456
24
065
```

2. Hexadezimalzahlen

```
$FF
$1F
$0080
```

3. Oktalzahlen

```
@70@55@012
```

4. Binärzahlen

```
%10101010
%01100110
```

5. Symbole

```
.Start
_LVOOpenLibrary
__RS
```

3.2.8 Operationen und deren Prioritäten

Pri	Operation		
7	-	negatives Vorzeichen	z.B. `-4, -$F0`
7	~	NOT	z.B. `~1, ~%0010`
6	*	Multiplikation	z.B. `LN_SIZE*4`
6	/	Division	z.B. `640/2`
6	%%	Modulo (Rest aus Division)	z.B. `100%%16`
5	+	Addition	z.B. `Start+4`
5	-	Subtraktion	z.B. `CNT-1`

4	<<	nach links schieben	z.B. `7<<5`
4	>>	nach rechts schieben	z.B. `2>>CNT`
3	&	Logisches UND	z.B. `XY&$7F`
2	^	Logisches EOR	z.B. `XY^$7F`
1	|	Logisches ODER	z.B. `GADGETUP|GADGETDOWN`
1	!	Logisches ODER	z.B. `GADGETUP!GADGETDOWN`

Beispiele:

```
(LN_SIZE>>2)-1
2*(4+5)              ; gibt 18
2*4+5                ; gibt 13
```

3.3 Macros

Macros sind häufig benötigte Befehle oder Befehlsfolgen, die zu einer Einheit zusammengefaßt werden. Ein Macro ist also nichts weiter, als ein neuer Befehl, den Sie definieren und der aus einem oder mehreren Kommandos besteht. Stellen Sie sich vor, Sie wollen die DOS-, Intuition- und Graphics-Library öffnen. Die folgenden Befehle

```
lea Name,a1
moveq   #Lib_Version,d0
jsr _LVOOpenLibrary(a6)
move.l  d0,LibBase
beq Exit
```

müßten Sie also dreimal schreiben. Kürzer und übersichtlicher gestaltet sich dagegen der Einsatz eines Macros:

```
* Macrodefinition

OpenLib     MACRO
lea \1Name,a1
moveq   #Lib_Version,d0
jsr _LVOOpenLibrary(a6)
move.l  d0,\1Base
beq Exit
ENDM
```

Das Macro und somit Ihr neuer Befehl heißt „OpenLib". Das Öffnen aller Libraries sieht dann folgendermaßen aus:

```
* Macroaufruf

OpenLib Dos
OpenLib Int
OpenLib Gfx
...
...
```

```
DosBase     dc.l    0
IntBase     dc.l    0
GfxBase     dc.l    0
DosName     dc.b    "dos.library",0
IntName     dc.b    "intuition.library",0
GfxName     dc.b    "graphics.library",0
```

Trifft der Assembler auf den neuen Befehl „OpenLib", so assembliert er nichts anderes, als die Zeilen, die in der Definition zwischen MACRO und ENDM stehen.

Dem Macro können natürlich auch Parameter übergeben werden. Im obigen Fall übergeben Sie `"Dos"`, `"Int"` bzw. `"Gfx"`. Trifft der Assembler bei der Bearbeitung eines Macros auf die Zeichen `\n`, so ersetzt er diese Zeichenfolge durch den entsprechenden Parameter. n ist die laufende Nummer des Parameters.

Zeile 1 des Macros:

```
lea \1Name,a1
```

wird beim ersten Ausruf also zu:

```
lea DosName,a1
```

Einem Macro dürfen Sie maximal 36 Parameter übergeben, die durch Kommata getrennt werden. Die Parameter werden im Macro mit `\1` bis `\9` und `\A` bis `\Z` bezeichnet. `\A` steht dabei für Parameter Nr. 10 und `\Z` für Nr. 36.

Macros dürfen beliebig lang sein, d.h. die maximale Länge wird nur durch den zur Verfügung stehenden Arbeitsspeicher begrenzt.

Macros dürfen auch ineinander geschachtelt werden, d.h. aus einem Macro heraus kann ein weiteres aufgerufen werden, aus diesem wiederum ein drittes usw. Die maximale Schachteltiefe ist unbegrenzt.

```
CALL    MACRO
jsr _LVO\1(a6)
ENDM

LINKLIB     MACRO
move.l  a6,-(sp)
move.l  \2Base,a6
CALL    \1              ;aus diesem Macro wird ein
                        ;weiteres aufgerufen
move.l  (sp)+,a6
ENDM
```

Eine Macrodefinition beginnt also mit der Direktive **`MACRO`**. In derselben Zeile MUSS der Name des Macros stehen. Alle nun folgenden Zeilen werden in die Macrodefinition übernommen, bis die Direktive **`ENDM`** erreicht wird.

In Macros dürfen natürlich auch Variablen definiert werden. Dabei spielt es keine Rolle, ob es sich um globale oder lokale Variablen handelt. Es kommt jedoch zu Problemen, wenn in einem Macro eine globale Variable definiert wird und Sie dieses Macro mehrmals benutzen. In diesem Fall würde der Assembler versuchen, die Variable mehrfach zu definieren, was insbesondere bei Labeln zu einer Fehlermeldung führen würde. Das gleiche Problem tritt auf, wenn Macros mit Definitionen lokaler Variablen mehrfach aufgerufen werden, ohne daß eine globale Variable definiert wird. Aus diesem Grund gibt es Variablen, die nur innerhalb eines Macros gelten (siehe auch 3.2.2). Sie beginnen mit einem Buchstaben oder Underscore (`"_"`) und enden mit der Kennung `"\@"`.

```
ABS     MACRO
        tst.l   \1
        bpl.s   NotNeg\@
        neg.l   \1
NotNeg\@
        ENDM
```

Dieses Macro ermittelt den Betrag einer Zahl. Das Label `NotNeg\@` gilt nur innerhalb des Macros.

Mit der Assembler-Version 5.0 stehen eine Reihe neuer Macrofunktionen zur Verfügung.

`\@` wird durch „_nnn" ersetzt, wobei nnn die laufende Nummer des Macros ist.
z.B. wird `Loop\@` zu `Loop_4`, wenn der interne Macrozähler auf 4 stand. Die laufende Nummer wird dezimal eingetragen.

`\#` wirkt wie `\@`, nur wird vor nnn kein `"_"` gesetzt. z.B.: aus `0\#$` wird `04$`

`\0-` enthält die Operandenlänge, die beim Macroaufruf benutzt wurde :

```
Test  MACRO
   tst.\0   \1
   ENDM
```

`Test d0` wird zu: `tst.w d0`

`Test.b (sp)` wird zu : `tst.b (sp)`

Schlüsselwörter/Funktionen

Alle folgenden Funktionen werden mit `"\*"` eingeleitet. Das Argument muß in Klammern stehen. Im Argument dürfen weitere Funktionen vorkommen, es darf maximal 16-fach geschachtelt werden.

*1. *VALOF(<Symbol>)*

Die Funktion berechnet den numerischen Wert ihres Argumentes (Symbol) und trägt diesen Wert als Dezimalzahl in die Macrozeile ein:

```
Exec = 4

Test  MACRO
   move.l   \*VALOF(Exec).w,a6
  ENDM
```

```
Test
```
wird zu:

```
move.l 4.w,a6
```

2. *STRLEN(<String>)

Diese Funktion berechnet die Anzahl der Zeichen des Argumentes und trägt die Länge als Dezimalzahl ein.

z.B.:

```
Test  MACRO
   dc.b  \*STRLEN(\1)
   dc.b  "\1"
   ENDM

   Test  <Example>
```
wird zu:

```
   dc.b  7
   dc.b  "Example"
```

3. *LEFT(<String>,n)

Diese Funktion trägt die ersten n Zeichen des Strings in die Macrozeile ein.

```
Test  MACRO
   dc.b  "\*LEFT("\1",4)"
   ENDM

Test  <Example>
```
wird zu:

```
dc.b  "Exam"      ; die ersten 4 Zeichen
```

4. *RIGHT(<String>,n)

Diese Funktion trägt die letzten n Zeichen des Strings in die Macrozeile ein.

5. *MID(<String>,s,n)

Diese Funktion übernimmt ab Position s aus dem String n Zeichen und trägt diese in die Macrozeile ein.

z.B.:

```
Test  MACRO
   dc.b  "\*MID("\1",4,2)"
   ENDM

Test  <Example>
```
wird zu :

```
dc.b  "mp"  ; ab dem 4.Zeichen 2 Zeichen übernehmen
```

6. *UPPER(<String>)

Diese Funktion wandelt alle Buchstaben des Strings in Großbuchstaben um.

```
Test  MACRO
   dc.b  "\*UPPER("\1")"
   ENDM

Test  <Example>
```

wird zu:

```
dc.b  "EXAMPLE"
```

3.4 Bedingte Assemblierung

Der Assembler ermöglicht Ihnen, bestimmte Programmteile in Abhängigkeit einer Bedingung zu assemblieren. Anwendungsgebiete dafür sind z.B. das Assemblieren von Include-Dateien in nur einem Pass oder die wiederholte Assemblierung mit rekursiven Macros. Der Aufbau einer IF-Konstruktion gestaltet sich wie folgt:

```
IFcc    <Term1>[,<Term2>]
...             ;Befehle, die assembliert werden,
                ;wenn das Ergebnis WAHR war
ELSE            ; optional !
...             ;Befehle, die assembliert werden,
                ;wenn das Ergebnis FALSCH war
ENDIF
```

Für cc sind alle Conditioncodes erlaubt, die Sie von den bedingten Sprungbefehlen des MC 68000 her kennen.

```
    IFEQ    1,2     ;ist immer falsch !
    IFEQ    1-1     ;ist immer wahr !
    IFCS    1,2     ;ist falsch, da 2 nicht größer als 1 ist

CALL   MACRO
    IFEQ    NARG-2  ;wurden genau 2 Parameter übergeben?
    move.l \2,a6    ;ja,dann diesen Befehl assemblieren!
    ELSE
    move.l 4.w,a6   ;sonst diesen
    ENDIF
    jsr \1(a6)
    ENDM

CALL    -30,DosBase
```

führt also zu:

```
   move.l  DosBase,a6
   jsr -30(a6)
```

während

```
CALL    -30
```

zu

```
    move.l  4.w,a6
    jsr -30(a6)
```

wird.

Auch wiederholte Assemblierung in Verbindung mit rekursiven Macros ist sehr leicht möglich:

```
TABLE  MACRO
    dc.b    CNT
```

```
CNT=CNT+1
        IFCC    CNT,100
    TABLE       ;ruft sich solange auf, bis CNT > 100!

    ENDIF
    ENDM

CNT=1
    TABLE       ;erzeugt eine Tabelle aus Bytes von
                ;1 bis 100
CNT=50
    TABLE       ;erzeugt eine Tabelle aus Bytes von
                ;50 bis 100
```

Weiterhin gibt es noch einige zusätzliche Direktiven zum bedingten Assemblieren:

```
IFC <String1>,<String2>
```

vergleicht zwei Strings und ist TRUE, wenn die Strings gleich sind. Es wird nicht zwischen Groß- und Kleinschreibung unterschieden, beide Strings müssen gleich lang sein.

```
IFNC <String1>,<String2>
```

ist FALSE, wenn Strings gleich sind und die gleiche Länge haben.

```
IFD  <Variable>
```

ist TRUE, wenn die Variable definiert ist.

```
IFND <Variable>
```

ist FALSE, wenn die Variable bereits definiert ist.

3.5 ALIGNment

Insbesondere bei Opcodes ist es zwingend notwendig, daß sie an einer geraden Adresse beginnen. Auch Datenbereiche, die Wort- oder Langwortdaten enthalten, sollten an geraden Adressen beginnen. Zudem verlangen viele DOS-Funktionen, daß Strukturen an durch 4 teilbaren Adressen beginnen.

Um dies sicherzustellen, existieren folgende Direktiven:

```
CNOP    <D>,<A>
```

`CNOP` bewirkt, daß die Adresse durch `<A>` teilbar gemacht wird. Danach wird noch `<D>` addiert, so daß Sie auch erreichen können, daß z.B. zwei Bytes hinter einem ALIGNment weiterassembliert wird. Eventuelle Füllbytes werden mit `NULL` gefüllt.

```
CNOP 0,2        ; der nächste Befehl steht an einer geraden ; Adresse
CNOP 0,4        ; der nächste Befehl steht an einer durch
                ; 4 teilbaren Adresse
```

```
CNOP 2,4          ; der nächste Befehl steht 2 Bytes hinter
                  ; einer durch 4 teilbaren Adresse
```

Höhere ALIGNments (größer 8) sind nicht sinnvoll, da die Startadresse eines Programmes im Speicher maximal durch 8 teilbar sein kann.

```
ALIGN.W       ; hat die gleiche Wirkung wie CNOP 0,2
ALIGN.L       ; wie CNOP 0,4
EVEN          ; wie ALIGN.W (gerade Adresse)
```

Ab dem MC 68020 dürfen Wort- und Langwortdaten auch an ungeraden Adressen stehen (aber keine Opcodes !). Sie können deshalb Fehlermeldungen bei Worten bzw. Langworten an ungeraden Adressen unterbinden:

```
ODDOK     ; erlaubt Worte und Langworte an ungeraden Adressen
ODDERR    ; gibt die Fehlermeldung bei ungeraden Adressen
          ; wieder frei (voreingestellt)
```

Opcodes an ungeraden Adressen führen aber auch bei **ODDOK** zu einer Fehlermeldung!

3.6 Sektionierung

Programme können in Sektionen, sogenannte Hunks zerlegt werden. Beim Laden des Programmes werden diese Sektionen dann im Speicher verteilt und es muß somit kein großer Speicherblock für das Gesamtprogramm frei sein. Außerdem gibt es sogenannte BSS-Sektionen, die uninitialisierte Daten (NULL) enthalten und nicht mit in der Objektdatei abgespeichert werden. Ihr Programm darf maximal 256 Sektionen enthalten. Der Standardbefehl zur Einleitung einer neuen Sektion heißt:

```
SECTION <Name>,<Typ>[,<Memory>]SECTION <Name>,<Typ>[_<Mem>]
```

Der **<Name>** wird nur berücksichtigt, wenn Sie ein Link-File erzeugen möchten. Es gibt drei Sektionstypen:

CODE	für den Programmcode oder für Daten
DATA	für vorinitialisierte Daten (z.B. Texte)
BSS	für Daten, die erst während des Programmlaufs anfallen
TEXT	hat die gleiche Wirkung wie CODE.
Optional	ist die Angabe des Speichertyps. Erlaubt sind:
CHIP bzw. _C	CHIP-RAM
FAST bzw. **_F**	FAST-RAM
PUBLIC bzw. **_P**	jeder Speicher wird akzeptiert (voreingestellt)

```
SECTION "",DATA,CHIP
```
bzw.
```
SECTION "",DATA_C
```

erzeugt eine DATA-Sektion, die in das CHIP-RAM gelegt wird.

```
SECTION "",CODE,FAST
```

bzw.

```
SECTION "",CODE_F
```

erzeugt eine CODE-Sektion, die nach dem Laden im FAST-RAM liegen wird. Wohin der Assembler diese Sektion legt, ist vom zur Verfügung stehenden Speicher abhängig, FAST-RAM läßt sich also beim Assemblieren nicht erzwingen!

Um sich Tipparbeit zu sparen, können Sie auch folgende Schlüsselworte anstelle des SECTION-Befehles verwenden:

```
CODE    ; = SECTION "",CODE
TEXT    ; = SECTION "",CODE
DATA    ; = SECTION "",DATA
BSS     ; = SECTION "",BSS
```

Die Schlüsselworte **CODE**,**TEXT**,**DATA** und **BSS**, die Sie zur Einleitung einer neuen Sektion alternativ zum SECTION-Befehl verwenden konnten, existieren nicht mehr.

Gleichnamige Sektionen werden zusammengefaßt. Gleichnamige Sektionen unterschiedlichen Typs oder mit unterschiedlichen Speicherattributen erzeugen eine Fehlermeldung!

Achtung: Im Objektfile erhalten die Sektionen aus Ihrem Programm unter Umständen eine andere Reihenfolge. Bei der Erzeugung des Objektfiles werden die Programmsektionen nach dem Typ geordnet. Erst werden alle **CODE**-, dann die **DATA**- und zum Schluß die **BSS**-Sektionen ins Objektfile geschrieben. Leere Sektionen, auf die nicht zugegriffen wird, werden nicht mit ins Objektfile geschrieben!

Direktiven zur Steuerung der Code-Erzeugung

MC68000

Dieser Zustand ist voreingestellt, es wird Code erzeugt, der auf allen Prozessoren lauffähig ist. Alle Befehle des 68010,68020... sind verboten.

MC68010

Die zusätzlichen Befehle des MC 68010 werden erlaubt.

MC68020

Die zusätzlichen Befehle und Adressierungsarten des MC 68020 werden erlaubt.

MC68030

Wirkt wie MC68020, es werden aber auch die Befehle zur Programmierung der im 68030 integrierten MMU zugelassen.

MC68881 / MC68882

Mit diesen Direktiven werden die Befehle der FPU zugelassen. Möchten Sie die erweiterten Adressierungsarten (s.o.) nutzen, müssen Sie diese noch mit MC68020(MC68030) freigeben.

MC68851

Diese Direktive erlaubt die Programmierung der MC68851-MMU. Achten Sie bitte darauf, daß alle MMU-Befehle priviliegiert sind!

Die Direktive MC68000 macht alle obigen Freigaben wieder rückgängig. Benutzen Sie die Direktiven vorsichtig, es kann nämlich sein, daß auch wenn Sie keine Befehle oder Adressierungsarten z.B. des MC68020 nutzen, Code für den MC68020 entsteht:

```
   MC68000
   move.l   (Test,PC),d0

   MC 68020
   move.l   (Test,PC),d0
   rts

Test  dc.l  0
```

Die beiden gleichen Befehle ergeben unterschiedlichen Code! Beide enthalten eine Vorwärtsreferenz, im zweiten wird ein 32-Bit-Displacement erzeugt!

3.7 Erzeugung von absolutem Code

Mit der Direktive ORG können Sie die Adresse angeben, ab der ein Programm lauffähig sein soll. Nach dem Auftreten von ORG werden für absolute Adressierungen keine Reloc-Informationen angelegt! Vor ORG dürfen keine codeproduzierenden Direktiven stehen, ORG sollte direkt auf einen Sektionsbefehl folgen.

Alternativen zu ORG:

```
ORIGIN
RORG
RORIGIN
```

z.B.:

```
   Section  "Test",Code

   ORG   $40000

L1 lea   L1,a0    ; eine absolute Adressierung, kein Reloc!
   rts
```

`L1` hat den Wert `$40000`, nicht `NULL` wie bei relozierbaren Programmem !

3.8 Konstanten

Um Daten in Ihrem Programm abzulegen, verwenden Sie die Direktive

```
DC.x  <Konstante>[,<Konstante>,...]
```

`x` gibt dabei die Länge an und kann B für Byte-Konstanten, `W` für Wort-Konstanten bzw. `L` für Langwort-Konstanten sein. Mehrere Konstanten werden dabei durch Kommata getrennt. Die Konstante darf ein mathematischer Ausdruck, eine Variable oder ein ASCII-Text, der in Hochkommata bzw. Anführungszeichen eingeschlossen sein muß, sein.

```
dc.b  0,1,2,3,4,5,6,7,8
```
wird assembliert zu:

```
00 01 02 03 04 05 06 07 08
```

```
dc.w  0,1,2,3,4
```
wird assembliert zu:

```
00 01 00 02 00 03 00 04
```

```
dc.b  "dos.library",0
```
wird assembliert zu:

```
64 6F 73 2E 6C 69 62 72 61 72 79 00
```

Für die Ablage von Texten gibt es noch zwei spezielle Direktiven, die einige Schreibarbeit sparen können:

CSTRING <String>[,<String>,...]

Legt den/die String(s) byteweise im Speicher ab und fügt ein NULL-Byte an.

```
CSTRING dos.library
```

hat die gleiche Funktion wie:

```
dc.b "dos.library",0
```

PSTRING <String>[,<String>,...]

Legt den/die String(s) byteweise im Speicher ab. Das erste Byte ist hierbei jedoch nicht das erste Zeichen, sondern enthält die Länge des folgenden Strings:

```
PSTRING DF0
```
wird assembliert zu:

```
03 44 46 30 3A
   ^ ^ Länge des folgenden Strings (3 Byte)
```

Für die DC.x-Direktive existieren einige Alternativen:

```
DB für DC.B
DW für DC.W
DL für DC.L
```

Die Direktive DC erlaubt nun auch die Angabe der Operandenlänge der FPU-Befehle.
Vorher sollten Sie jedoch die Erzeugung von FPU-Codes aktivieren. Hier eine Aufstellung aller Möglichkeiten:

`DC.B`	Bytes
`DC.W`	Words
`DC.L`	Longwords
`DC.S`	IEEE-Single, ein Langwort
`DC.D`	IEEE-Double, zwei Langworte
`DC.X`	IEEE-Extended, drei Langworte
`DC.P`	decimal packed, drei Langworte (nur mit FPU möglich !)

Format der Fließkomma-Ausdrücke:

```
[+|-]xxx.yyy[E[+|-]zzz]
```

z.B.:

```
     3.14e3
-50.10e20
50e-11
     3.1415927*2.5
```

Beispiel :

```
DC.X  3.1415927
```

legt 3.1415927 im IEEE-Extended-Format (3 Langworte) in den Speicher

```
DC.S  3.1415927
```

legt 3.1415927 im IEEE-Single-Format (3 Langworte) in den Speicher

Kommen in Ihrem Programm Fließkomma-Konstanten vor, müssen auf der Boot-Disk im LIBS-Verzeichnis die mathieeedoubbas.library und die mathieeedoubtrans.library vorhanden sein. Diese Libraries werden zur Erzeugung der verschiedenen FPU-Zahlenformate verwendet und arbeiten automatisch mit einer eventuell vorhandenen FPU.

3.9 Reservierte Speicherblöcke

DS.x <Größe>[,<Füllwert>]

Mit dieser Direktive wird ein Block definiert, der mit `<Füllwert>` initialisiert werden kann. Für `x` sind wieder `B` (Bytes), `W` (Words) und `L` (Langworte) erlaubt.

```
DS.B  10,-1
```

definiert einen Block aus 10 Bytes, die mit $FF initialisiert werden.

```
DS.L  3
```

hat die gleiche Funktion wie:

```
DC.L  0,0,0
```

Der Sonderfall `DS.W 0` bringt die Adresse wieder auf Wortgrenze, hat damit also die gleiche Bedeutung wie `CNOP 0,2`. Analog dazu macht `DS.L 0` die Adresse durch 4 teilbar (`CNOP 0,4`). An Stelle von `DS` können Sie auch `BLK` oder `DCB` schreiben.

Auch bei der Direktive `DS` (und ihren Alternativen) ist die Angabe von FPU-Operandenlängen möglich.

```
DS.X  40
```

reserviert Platz für 40 Extended-Zahlen (40*3=120 Langworte)

3.10 Definition einer Variablen

Variablen sind wie unter 3.2.2. beschrieben grundsätzlich redefinierbar. Mit der Direktive EQUwird einer Variablen ein Wert zugewiesen. Der Name der Variablen muß in der ersten Spalte beginnen. Der Wert der Variable muß ein mathematischer Ausdruck sein.

```
Exec    EQU 4
```

weist der Variablen `"Exec"` den Wert 4 zu.

```
CNT     EQU CNT+1
```

erhöht den Wert der Variablen `"CNT"` um 1.

Anstelle von `EQU` können Sie auch `"="`, `"=="` oder `"SET"` schreiben:

```
Exec    =   4
Exec    ==  4
Exec    SET 4
```

3.11 Definition von Strukturoffsets

Zum Erstellen eigener Strukturen oder zur Definition der Strukturen des Betriebssystems stellt Ihnen der Assembler einen internen Zähler zur Verfügung. Ein Feld einer Struktur wird wie folgt definiert:

```
<Name> RS.x     <Anzahl>
```

x kann B für Bytes, W für Words und L für Langworte annehmen. Die Anzahl bestimmt, aus wievielen Bytes (Words oder Langworten) das Feld besteht. Die RS-Direktive weist der Variablen den aktuellen RS-Zähler zu und erhöht diesen in Abhängigkeit der Größe x und der Anzahl.

```
RSSET <Wert>
```

Setzt den internen Zähler auf einen bestimmten Anfangswert.

```
RSRESET
```

Setzt den internen Zähler auf 0. Der aktuelle Wert dieses Zählers läßt sich der Variable `__RS` entnehmen (siehe 3.2.2.5.2).

```
RSRESET                 ;löscht den Zähler
XYZ     RS.L    1       ;weist der Variablen XYZ den Wert 0 zu
                        ;und erhöht __RS um 4 (ein Langwort)
```

Definieren Sie Worte oder Langworte, wird geprüft, ob der Zähler ungerade ist. Wenn ja, wird er vorher um eins erhöht, so daß Words und Langworte immer gerade Offsets erhalten !

```
RSRESET                 ; __RS = 0
XY      RS.B    1       ; __RS = 1
XZ      RS.L    1       ; XZ enthält den Wert 2 !
                        ; __RS = 6
```

3.12 Externe Quelltexte

Der Assembler unterstützt die Möglichkeit, Quelltexte von externen Speichermedien in das Hauptprogramm einzubinden.

```
INCLUDE <Datei>
```

Leitet an dieser Stelle die Assemblierung in die angegebene Datei um. Include-Dateien dürfen beliebig tief geschachtelt werden und beliebig lang sein. Es existieren jedoch einige Konventionen bezüglich des Gebrauchs von Include-Dateien:

- IF-Konstruktionen, die in der Datei geöffnet werden, müssen in der gleichen Datei auch wieder geschlossen werden.
- Macrodefinitionen müssen in der gleichen Datei beendet sein, in der sie begonnen wurden.
- Include-Dateien, die Programm (Code) enthalten, müssen in allen Passes INCLUDEt werden.

Beispiele:

```
INCLUDE exec/types.i
INCLUDE exec/node.i
```

Der Assembler sucht normalerweise im Hauptverzeichnis der Boot-Disk nach den Include-Dateien. Um dem Assembler weitere Verzeichnisse bekannt zu machen, verwenden Sie bitte den Befehl:

```
INCDIR <Pfad>
```

Fügt den angegebenen Pfad in die Suchliste der Include-Dateien ein. Bis zu maximal 32 Pfade können mit diesem Befehl angemeldet werden.

```
INCDIR  DF0:
INCDIR  DF1:
INCDIR  DF1:Includes
```

bewirkt, daß der Assembler in diesen drei Verzeichnissen nach einer zu INCLUDEnden Datei sucht.

3.13 Externe Rohdateien

Möchten Sie z.B. Graphik- oder Sounddaten in Ihre Programme einbinden, und sind diese zu umfangreich, um sie in `DC.x`-Zeilen zu konvertieren, können Sie die Daten direkt von Disk in den Programmcode einbinden.

```
INCBIN <Datei>[,<Länge>]
```

Die angegebene Datei wird direkt in den Programmcode integriert. Die Längenangabe ist optional und gibt an, wieviele Bytes der Datei geladen werden sollen. Geben Sie die Länge nicht an, oder ist sie größer als die Dateilänge, so wird die Datei komplett geladen. Nach dem Laden wird die Adresse nicht wieder auf Wortgrenze gebracht, d.h. Sie müssen selbst darauf achten, daß folgende Codes an geraden Adressen beginnen.

3.14 Copper-Assembler

Der Assembler unterstützt die direkte Programmierung von Copperbefehlslisten. Der Copper ist einer der Coprozessoren des Amiga und steuert wichtige Parameter der Bildausgabe in Abhängigkeit der aktuellen Rasterstrahlposition. Der Befehlssatz des Coppers ist zwar nicht besonders umfangreich, er umfaßt nur drei Befehle, die aber ausreichen, sogar Schleifen oder komplexe sich selbst modifizierende Befehlslisten zu programmieren.

Die Befehle im einzelnen sind:

`CMOVE` ; Beschreiben eines Registers

`CWAIT` ; auf eine bestimmte Position warten

`CSKIP` ; testen, ob die angegebene Position schon erreicht war und wenn ja, den nächsten Befehl übergehen

In einer Copperliste hat jeder Befehl eine feste Länge von 4 Bytes. Das Bit 0 beider Befehlsworte dient zum eindeutigen Erkennen der einzelnen Befehle:

erstes Wort	zweites Wort	
0	x	CMOVE (x = 0|1)
1	0	CWAIT
1	1	CSKIP

Aufbau der Befehle im einzelnen:

Bit	CMOVE		CWAIT		CSKIP	
	1.Wort	2.Wort	1.Wort	2.Wort	1.Wort	2.Wort
15	RB15	DT15	VP7	BFD	VP7	BFD
14	RB14	DT14	VP6	VM6	VP6	VM6
13	RB13	DT13	VP5	VM5	VP5	VM5
12	RB12	DT12	VP4	VM4	VP4	VM4
11	RB11	DT11	VP3	VM3	VP3	VM3
10	RB10	DT10	VP2	VM2	VP2	VM2
9	RB9	DT9	VP1	VM1	VP1	VM1
8	RB8	DT8	VP0	VM0	VP0	VM0
7	RB7	DT7	HP8	HM8	HP8	HM8
6	RB6	DT6	HP7	HM7	HP7	HM7
5	RB5	DT5	HP6	HM6	HP6	HM6
4	RB4	DT4	HP5	HM5	HP5	HM5
3	RB3	DT3	HP4	HM4	HP4	HM4
2	RB2	DT2	HP3	HM3	HP3	HM3
1	RB1	DT1	HP2	HM2	HP2	HM2
0	0	DT0	1	0	1	1

`RB15...RB0:` Offset des Zielregisters (z.B.: $0096 für DMACON)
`DT15...DT0:` Datenwort, das in das Register übertragen wird
`VP7 ...VP0:` vertikale Position des Rasterstrahls
`HP8 ...HP1:` horizontale Rasterstrahlposition
`VM7 ...VM0:` Maske für vertikale Rasterstrahlposition
`HM8 ...HM1:` Maske für horizontale Rasterstrahlposition
`BFD :` Blitter-Finish-Disable Bit
(0 = auf Ende der Blitteroperation warten)

Wie Sie sehen, ist der Aufbau der Befehle `CWAIT` und `CSKIP` bis auf das Bit 0 des zweiten Wortes identisch. Doch nun zur Syntax der einzelnen Befehle:

Es ist zwingend notwendig, daß Copper-Befehlslisten immer im CHIP-RAM stehen. Weiterhin müssen Copper-Befehle an geraden Adressen stehen!

```
CMOVE  #<Data>,<Register>  oder  CMOVE #<Data>,(<Register>)
```

<Data> ist das Datenwort, welches in das entsprechende Register übertragen werden soll.

```
CMOVE #$0000,$180       ; setzt die Hintergrundfarbe auf schwarz
CMOVE #$7FFF,$096       ; schaltet alle DMA's aus ($096 = DMACON)
                        ; Vorsicht! Danach funktioniert im Prinzip
                        ; nichts mehr! (kein Bild, kein Ton, kein
                        ; Diskzugriff ...)
```

```
CLMOVE #<Data>,<Register>  oder  CLMOVE #<Data>,(Register)
```

<Data> ist hierbei ein Langwort, das auf zwei Register verteilt wird. **CLMOVE** erzeugt also zwei einzelne **CMOVE**-Befehle. Das obere Wort von <Data> wird in das angegebene Register übertragen und das untere Wort in <Register>+2.

```
CLMOVE #0,$180          ;setzt die Hintergrundfarbe und Farbe
                        ;#1 auf schwarz (0)
```

```
CWAIT  <VPos>,<HPos>[{<VMask>,<HMask>}BFD]
```

VPos: vertikale Position von $00 bis $FF
HPos: horizontale Position von $02 bis $1FE
VMask: Maske für Vertikalposition
HMask: Maske für Horizontalposition
BFD : BFD-Bit 0 | 1

```
CWAIT $2c,0{}0
```

Wartet auf die Position **$2C** (Vert.) 0 (Horiz.) und auf das Ende der letzten Blitteroperation (**BFD=0**). Als Maske wird voreingestellt **$7F** (Vert.) und **$1FE** (Horiz.) eingesetzt.

```
CWAIT $2c,0
```

Wartet auf Postition **$2C**,0 jedoch nicht auf das Ende der letzten Blitteroperation.

Die Syntax des **CSKIP**-Befehles ist identisch mit der des **CWAIT**-Befehles. Haben Sie bemerkt, daß es keinen **END**-Befehl zum Abschluß einer Copper-Liste gibt? Um eine Copperliste zu beenden, verwendet man einen **CWAIT**-Befehl, der auf eine unmögliche Rasterposition wartet.

Z.B.:

```
CWAIT  $FF,$1FE
```

Die horizontale Position **$1FE** existiert nicht, und der Copper wartet sozusagen umsonst. Ist das Bild fertig dargestellt, wird jedoch die Copperliste neu gestartet, so daß der Copper sich nicht „aufhängen" kann. Um die Copperliste zu beenden, können Sie aber auch die folgende Direktive verwenden:

```
CEND
```

CEND erzeugt nichts anderes, als den oben beschriebenen unmöglichen **CWAIT**-Befehl.

Ein Problem ergibt sich allerdings noch aus dem Aufbau der **CWAIT**- und **CSKIP**-Befehle. Für die Angabe der vertikalen Position stehen nur 8 Bits (**VP0...VP7**) zur Verfügung. Ein PAL-Bild hat aber mehr als 256 Zeilen. Die übrigen Zeilen erreichen Sie, wenn Sie auf das Ende der Zeile 255 warten. Die übrigen Zeilen sind dann wieder ab Zeile 0 ... erreichbar.

Copperbefehle müssen mit der Direktive **COPPER** erst zugelassen werden. Die Erzeugung von Copper-Befehlen kann mit **ENDCOP** wieder verboten werden.

z.B.:

```
Color00=$180
Color01=$182
Color02=$184
Color03=$186

   COPPER

   CMOVE #$AAA,Color00
   CMOVE #$000,Color01
   CMOVE #$FFF,Color02
   CMOVE #$68B,Color03
   CEND

   ENDCOP
```

3.15 Externe Symbole

Um dem Assembler bzw. dem Linker externe Symbole bekannt zu machen, existieren folgende Direktiven:

```
XREF <Symbol1>[,<Symbol2>...]
```

Diese Direktive teilt dem Assembler mit, daß die angegebenen Symbole nicht in diesem Codemodul definiert werden, sondern aus einem anderen Modul importiert werden. Der Linker setzt dann an die Stellen, an denen dieses Symbol benutzt wird, die richtigen Werte ein. Diese Direktive dürfen Sie also nur benutzen, wenn Sie linkfähigen Code erzeugen möchten. Es werden 32-Bit, 16-Bit und 8-Bit Referenzen auf ein externes Symbol unterstützt.

Beispiel für eine 32-Bit-Referenz:

```
XREF    Main
Start   jsr Main          ;32-Bit-Referenz
...
```

Beispiel für eine 16-Bit-Referenz:

```
XREF    _LVOOpenLibrary
...
    jsr _LVOOpenLibrary(a6) ;16-Bit-Referenz
...
```

Beispiel für eine 8-Bit-Referenz:

```
XREF    LN_SIZE
...
moveq   #LN_SIZE,d0       ;8-Bit-Referenz
...
```

XREF führt zu einer Fehlermeldung, wenn Sie ausführbaren Code erzeugen möchten.

```
XDEF    <Symbol1>[,<Symbol2>...]
```

Exportiert die angegebenen Symbole, so daß sie von anderen Codemodulen benutzt werden können. Dabei wird zwischen relocatiblen Labeln und Variablen unterschieden.

```
DEF  Var     ;diese Variable wird anderen Modulen
             ;bekannt gemacht
XDEF _Main   ;_Main wird anderen Modulen bekannt
             ;gemacht
Var EQU 267
_Main   moveq   #0,d0
...
...
rts
```

Zusätzlich zu den oben benutzten Direktiven können Sie auch folgende Alternativen benutzen:

1. für **XREF:** EXTERN
2. für **XDEF:** PUBLIC
   ENTRY
   GLOBAL
   GLOBL

3.16 Codearten

Der Assembler unterscheidet zwischen drei verschiedenen Code-Arten:

1. Lauffähiger Code

Diesen Code werden Sie im Zusammenhang mit dem C-Compiler eher selten erzeugen. Immer, wenn Sie die Entscheidung über die Art des erzeugten Codes dem Assembler überlassen und die Direktive XREF bzw. deren Alternativen nicht benutzen, wird lauffähiger Code erzeugt.

2. Linkbarer Code (Option l)

Verwenden Sie in Ihrem Quelltext externe Referenzen (XREF), wird beim Abspeichern ein Link-File erzeugt, das Sie mit anderen Modulen zu einem lauffähigen Programm linken können. Linkfiles sind weder vom CLI noch von der Workbench startbar. Große Programmprojekte werden sinnvollerweise in Module zerlegt, die erst durch den Linker zu einem fertigen Programm zusammengefügt werden.

3. Absoluter Code (Option a)

Haben Sie ein lauffähiges Programm erzeugt, so können Sie hiermit absoluten Code abspeichern. Sie werden zu der Eingabe der Adresse aufgefordert, ab der das Programm später lauffähig sein soll. Absoluten Code können Sie allerdings weder vom CLI noch von der Workbench starten. Diese Option ist nur sinnvoll, wenn Sie von vornherein wissen, an welcher Adresse das Programm später laufen soll.

3.17 Assemblerbetriebsarten

Vor dem Assemblieren eines Quelltextes können Sie zwischen drei verschiedenen Betriebsarten wählen:

1. Normal

Der Assembler führt zwei Passes durch. Der Code wird nicht optimiert. Es entsteht Code, den Sie abspeichern können.

2. Optimieren (Option o)

Hiermit schalten Sie den Optimierungsmodus ein. Alle aktivierten Optimierungen werden versucht, es werden soviele Passes wie nötig durchgeführt, d.h. der Assembler erzeugt erst Code, wenn alle Optimierungsmöglichkeiten ausgeschöpft sind. Mindestens 3 Passes werden durchgeführt, nach oben sind im Prinzip keine Grenzen gesetzt. Die Anzahl der durchgeführten Passes hängt vom Programm ab. Es kann also durchaus vorkommen, daß der Assembler 16 Passes assembliert. Sie erhalten allerdings ein in Länge und Laufzeit hochoptimiertes Programm, so daß sich die Wartezeit lohnt.

3.18 Mögliche Optimierungen

Haben Sie die Multi-Pass-Optimierung aktiviert,wird folgendes optimiert:

```
MOVE.L » MOVEQ
```

Alle `MOVE.L`-Befehle, deren Langwort-Konstante im Bereich `-$80<=K<=$7F` liegt und in ein Datenregister übernehmen, werden in einen `MOVEQ`-Befehl umgewandelt. Sie sparen dabei 4 Bytes und 8 Taktzyklen (ein `MOVE.L #K,Dn` benötig 12 Taktzyklen gegenüber 4 Taktzylken für den `MOVEQ`-Befehl).

```
20 3C 00 00 00 3C    MOVE.L #$3C,D0
```

wird optimiert zu:

```
70 3C                MOVEQ  #$3C,d0
```

```
CLR.L Dn » MOVEQ #0,Dn
```

Alle `CLR.L Dn`-Befehle werden in ein `MOVEQ #0,Dn` umgewandelt. Der Opcode wird dabei nicht kürzer, ein `MOVEQ` ist aber um 2 Taktzyklen schneller.

```
LABEL » LABEL(PC)
```

Adressieren Sie ein Label, das in der gleichen Sektion liegt und nicht weiter als 32768 Bytes vom Befehl entfernt ist absolut, wird die Adressierung wenn möglich nach PC-Relativ umgewandelt. Der Befehl wird um 2 Bytes kürzer, die Ausführungszeit verkürzt sich um 4 Taktzyklen und die Programmdatei auf Disk wird um 4 Bytes kürzer (Relocoffset).

```
MOVE.L DosBase,a6
```

wird zu

```
MOVE.L DosBase(PC),a6
```

aber:

```
MOVE.L d0,DosBase
```

kann nicht optimiert werden, da PC-relative Adressierung nur im Quelloperand möglich ist.

```
XXXX.L » XXXX.W
```

Absolut lange Adressierungen werden in absolut kurz gewandelt. Das ist natürlich nur möglich, wenn die Adresse zwischen `-$8000` und `$7FFF` liegt. Ein Beispiel für diese Optimierung ist das Holen der Exec-Basis:

```
MOVE.L 4,A6
```

wird zu:

```
MOVE.L 4.W,A6
```

Der Befehl wird um zwei Bytes kürzer und um 4 Taktzyklen schneller.

```
Bcc.L » Bcc.S
```

Alle Branch-Befehle, deren Ziel im Bereich von -$80 und $7F liegt, werden in einen Short-Branch gewandelt. Der Befehl wird um zwei Bytes kürzer und um 4 Taktzyklen schneller, wenn die Verzweigung nicht ausgeführt wurde (außer **BRA** und **BSR**).

```
00(An) » (An)
```

Adressierungen mit der Distanz 0 werden in die Adressierung **ARI** ohne Distanz umgewandelt. Sie sparen dadurch 2 Bytes und die Ausführung wird um 4 Taktzyklen beschleunigt.

```
ADD/SUB » ADDQ/SUBQ
```

Addieren Sie eine Konstante von 1 bis einschl. 8, wird der **ADD**-Befehl (**SUB**) in einen **ADDQ**-Befehl (**SUBQ**) optimiert. Sie sparen dabei 2 Bytes und der Befehl wird um 4 Taktzyklen schneller. Bei Langwortverarbeitung sparen Sie sogar 4 Bytes.

```
CMP #0,<ea> » TST <ea>
```

Befehle, in denen Sie einen Operanden mit 0 vergleichen, werden in einen **TST**-Befehl umgewandelt (außer **CMP #0,An**). Sie sparen 2 Bytes (4 bei Langwortverarbeitung) und 4 Taktzyklen.

```
Label » Label(BASEREG)
```

Absolute Adressierungen werden in relative zu einem Adressregister umgewandelt. Dazu müssen Sie dem Assembler bekanntgeben, welches Adressregister verwendet werden soll und welcher Wert sich in diesem Register befindet. Dazu verwenden Sie bitte den Befehl:

```
BASEREG An,<Wert>
```

z.B.:

```
BASEREG A5,Data
DataDosBase dc.l    0
Handle  dc.l    0
```

führt zu folgenden Optimierungen:

```
move.l  DosBase,a6
```

wird zu

```
MOVE.L (A5),a6
```

```
move.l  Handle,d1
```

wird zu

```
MOVE.L 4(A5),a6
```

Diese Optimierung können Sie mit

```
BASEREG OFF
```

abschalten.

Das ist besonders wichtig, wenn Sie das Register, mit dem optimiert werden soll, initialisieren:

```
BASEREG OFF          ;abschalten
lea Data,a5          ;A5 initialisieren
BASEREG A5,Data      ;ab hier darf optimiert werden
```

3.19 Ausgabedateien

Das Protokoll einer laufenden Assemblierung kann in eine Datei gschrieben werden. Dazu haben Sie drei Möglichkeiten:

1. Listing (-l)

Es wird ein Assemblerlisting erzeugt, das die Quelltextzeile und den entstandenen Code auflistet.

```
ZEILE    PC     CODE                TEXT
    1 00000000 7000                Start moveq #0,d0
    2 00000002 7200                 moveq #0,d1
    3 00000004 41F90000000C         lea Label,a0
    4 0000000A 4E75                 rts
    5 0000000C FFFFFFFF            Label dc.l -1
```

Mit der Direktive NOLIST können Sie die Erzeugung eines Protokolls verbieten und mit LIST wieder erlauben.

Z.B.:

```
NOLIST
include exec/types.i     ;diese Include-Dateien erscheinen
include exec/nodes.i     ;nicht im Listing !
include exec/lists.i
LIST                     ;ab dieser Zeile wird wieder ein
                         ;Listing erzeugt.
```

2. Symboltabelle (-f)

Alle im Quelltext definierten globalen Symbole werden getrennt nach Labeln und Variablen in eine Datei ausgegeben.

3. Fehlerdatei (-u)

Sollten beim Assemblieren Fehler auftreten, werden diese nicht angezeigt, sondern in eine Datei umgeleitet. Die Assemblierung wird dann trotzdem fortgesetzt, so daß Sie alle Fehlermeldungen

erhalten. Es kann jedoch vorkommen, daß einige Fehler Folgefehler sind, da die Assemblierung einer Zeile beim Auftreten eines Fehlers abgebrochen wird.

Anhang A

Die Befehle des MC 68000 / MC 68010

Der folgende Überblick über die Befehle des MC 68000 / MC 68010 umfaßt für jeden Befehl die korrekte Syntax, eine kurze Beschreibung der ablaufenden Operation und der erlaubten Adressierungsarten.

Die verwendeten Abkürzungen haben folgende Bedeutung :

`Dn`	ein Datenregister D0 ... D7
`An`	ein Adressregitsre A0 ... A7
`Rn`	ein beliebiges Adress- oder Datenregister
`X`	das X-Flag
`N`	das N-Flag
`Z`	das Z-Flag
`C`	das C-Flag
`V`	das V-Flag
`<ea>`	effektive Adresse (eine der erlaubten Adressierungsarten)
`.x`	Operandengröße
`#K`	eine Konstante
`cc`	Condition-Codes
`d`	im Befehl angegebene relative Adressdistanz
`RList`	Registerliste z.B. d0-d7/a2-a3/a5

Operandengrößen in eckigen Klammern sind optional, brauchen also nicht angegeben zu werden (z.B. [B]: Der Befehl verarbeitet nur Byte-Operanden) Befehle vor denen ein `"*"` steht, sind nur im Supervisor-Mode erlaubt !

Adressierungsarten :

```
   #K  Konstante unmittelbar
d8(PC,Rn)  PC-Relativ mit Indexregister
  d16(PC)  PC-Relativ
      $.L  Absolut lang
      $.W  Absolut kurz
d8(An,Rn)  ARI mit Distanz und Indexregister
  d16(An)  ARI mit Distanz
    -(An)  ARI mit Predekrement
    (An)+  ARI mit Postinkrement
     (An)  Adressregister indirekt (ARI)
       An  Adressregister direkt
       Dn  Datenregister direkt
```

Befehl		Operation	Flags	000000000011 012345678901
ABCD Dn,Dn	[B]	Ziel=Ziel+Quelle-X BCD	XNZCV	
ABCD -(An),-(An)	[B]	Ziel=Ziel+Quelle-X BCD	XNZCV	
ADD.x Dn,<ea>	BWL	Ziel=Quelle+Ziel	XNZCV	..*******...
ADD.x <ea>,Dn	BWL	Ziel=Quelle+Ziel	XNZCV	**********
ADDA.x <ea>,An	WL	Ziel=Quelle+Ziel		**********
ADDI.x #K,<ea>	BWL	Ziel=Quelle+Ziel	XNZCV	*.*******...
ADDQ.x #K,<ea>	BWL	Ziel=Quelle+Ziel 0<K<9	XNZCV	*********...
ADDX.x Dn,Dn	BWL	Ziel=Quelle+Ziel+X	XNZCV	
ADDX.x -(An),-(An)	BWL	Ziel=Quelle+Ziel+X	XNZCV	
AND.x Dn,<ea>	BWL	Ziel=Quelle^Ziel	XNZCV	..*******...
AND.x <ea>,Dn	BWL	Ziel=Quelle^Ziel	XNZCV	**********
ANDI.x #K,<ea>	BWL	Ziel=Quelle^Ziel	XNZCV	*.*******...
ANDI #K,CCR	[B]	Ziel=Quelle^Ziel	XNZCV	
*ANDI #K,SR	[W]	Ziel=Quelle^Ziel	XNZCV	
ASL.x Dn,Dn	BWL	Ziel um Dn verschoben	XNZCV	
ASL.x #K,Dn	BWL	Ziel um #K verschoben	XNZCV	
ASL <ea>	[W]	Ziel um 1 verschoben	XNZCV	..*******...
ASR.x Dn,Dn	BWL	Ziel um Dn verschoben	XNZCV	
ASR.x #K,Dn	BWL	Ziel um #K verschoben	XNZCV	
ASR <ea>	[W]	Ziel um 1 verschoben	XNZCV	..*******...
Bcc Label		cc=TRUE : PC=PC+d		
BCHG Dn,<ea>		Bit testen und invert.	Z	*.*******...
BCHG #K,<ea>		Bit testen und invert.	Z	*.*******...
BCLR Dn,<ea>		Bit testen und löschen	Z	*.*******...
BCLR #K,<ea>		Bit testen und löschen	Z	*.*******...
BSET Dn,<ea>		Bit testen und setzen	Z	*.*******...
BSET #K,<ea>		Bit testen und setzen	Z	*.*******...
BTST Dn,<ea>		Bit testen	Z	*.*******...
BTST #K,<ea>		Bit testen	Z	*.*******...
BRA Label		PC=PC+d		
BSR Label		PC » Stack,PC=PC+d		
CHK <ea>,Dn	[W]	Dn<0|Dn><ea> » Except.	NZCV	***********
CLR.x <ea>	BWL	<ea>=0	NZCV	*.*******...
CMP.x <ea>,Dn	BWL	Flags=Quelle-Ziel	NZCV	**********

CMPA.x <ea>,An	WL	Flags=Quelle-Ziel	NZCV	**********
CMPI.x #K,<ea>	BWL	Flags=Quelle-Ziel	NZCV	*.******...
CMPM.x (An)+,(An)+	BWL	Flags=Quelle-Ziel	NZCV	
DBcc Dn,Label		cc=FALSE : Dn=Dn-1		
		Dn <> -1 : PC=PC+d		
DIVS <ea>,Dn	[W]	Ziel=Ziel/Quelle	NZCV	*.**********
DIVU <ea>,Dn	[W]	Ziel=Ziel/Quelle	NZCV	*.**********
EOR.x Dn,<ea>	BWL	Ziel=Ziel EOR Quelle	NZCV	*.*******...
EORI.x #K,<ea>	BWL	Ziel=Ziel EOR Quelle	NZCV	*.*******...
EORI.x #K,CCR	[B]	Ziel=Ziel EOR Quelle	XNZCV	
*EORI.x #K,SR	[W]	Ziel=Ziel EOR Quelle	XNZCV	
EXG Rn,Rn		Rn <-> Rn		
EXT.x Dn	WL	verzeichenr. erweitern	NZCV	
ILLEGAL		Illegal Opcode Except.		
JMP <ea>		PC=<ea>		..*..******.
JSR <ea>		PC » Stack,PC=<ea>		..*..******.
LEA <ea>,An		An=<ea>		..*..******.
LINK An,#D		An»Stack,SP»An,SP=SP+D		
LSL.x Dn,Dn	BWL	Ziel um Dn verschoben	XNZCV	
LSL.x #K,Dn	BWL	Ziel um #K verschoben	XNZCV	
LSL <ea>	[W]	Ziel um 1 verschoben	XNZCV	..*******...
LSR.x Dn,Dn	BWL	Ziel um Dn verschoben	XNZCV	
LSR.x #K,Dn	BWL	Ziel um #K verschoben	XNZCV	
LSR <ea>	[W]	Ziel um 1 verschoben	XNZCV	..*******...
MOVE.x <ea>,<ea>	BWL	Quelle » Ziel	NZCV	**********
				*.*******...
MOVE <ea>,CCR	[B]	CCR=<ea>	XNZCV	*.**********
*MOVE <ea>,SR	[W]	SR =<ea>	XNZCV	*.**********
MOVE SR,<ea>	[W]	<ea>=SR		*.*******...
*MOVE USP,An	[L]	An=USP		
*MOVE An,USP	[L]	USP=An		
MOVEA.x <ea>,An	WL	An=<ea>		**********
MOVEM.x RList,<ea>	WL	<ea>=RList		..*.*****...
MOVEM.x <ea>,RList	WL	RList=<ea>		..**.******.
MOVEP.x Dn,D16(An)	WL	Ziel=Dn		
MOVEP.x D16(An),Dn	WL	Dn=Quelle		
MOVEQ #K,Dn	[L]	Ziel=Quelle	NZCV	
MULS <ea>,Dn	[W]	Ziel=Ziel*Quelle	NZCV	*.**********
MULU <ea>,Dn	[W]	Ziel=Ziel*Quelle	NZCV	*.**********
NBCD <ea>	[B]	<ea>=0-<ea>-X	XNZCV	*.*******...
NEG.x <ea>	BWL	<ea>=0-<ea>	XNZCV	*.*******...
NEGX.x <ea>	BWL	<ea>=0-<ea>-X	XNZCV	*.*******...
NOP		Keine Operation		
NOT.x <ea>	BWL	<ea>=0-<ea>-1	NZCV	*.*******...
OR.x Dn,<ea>	BWL	Ziel=Ziel OR Quelle	NZCV	..******...
OR.x <ea>,Dn	BWL	Ziel=Ziel OR Quelle	NZCV	*.**********
ORI.x #K,<ea>	BWL	Ziel=Ziel OR Quelle	NZCV	*.*******...
ORI #K,CCR	[B]	Ziel=Ziel OR Quelle	XNZCV	
*ORI #K,SR	[W]	Ziel=Ziel OR Quelle	XNZCV	
PEA <ea>		<ea> » Stack		..*..******.
*RESET		Resetleitung LOW		
ROL.x Dn,Dn	BWL	Ziel um Dn verschoben	XNZCV	

ROL.x #K,Dn	BWL	Ziel um #K verschoben	XNZCV		
ROL <ea>	[W]	Ziel um 1 verschoben	XNZCV	..*******...	
ROR.x Dn,Dn	BWL	Ziel um Dn verschoben	XNZCV		
ROR.x #K,Dn	BWL	Ziel um #K verschoben	XNZCV		
ROR <ea>	[W]	Ziel um 1 verschoben	XNZCV	..*******...	
ROXL.x Dn,Dn	BWL	Ziel um Dn verschoben	XNZCV		
ROXL.x #K,Dn	BWL	Ziel um #K verschoben	XNZCV		
ROXL <ea>	[W]	Ziel um 1 verschoben	XNZCV	..*******...	
ROXR.x Dn,Dn	BWL	Ziel um Dn verschoben	XNZCV		
ROXR.x #K,Dn	BWL	Ziel um #K verschoben	XNZCV		
ROXR <ea>	[W]	Ziel um 1 verschoben	XNZCV	..*******...	
*RTE		PC und SR vom Stack	XNZCV		
RTR		PC und CCR vom Stack	XNZCV		
RTS		PC vom Stack			
SBCD Dn,Dn	[B]	Ziel=Ziel-Quelle-X BCD	XNZCV		
SBCD -(An),-(An)	[B]	Ziel=Ziel-Quelle-X BCD	XNZCV		
Scc <ea>		Setze <ea>,wenn cc=TRUE		*.*******...	
*STOP #K		SR=#K STOP-Zustand	XNZCV		
SUB.x Dn,<ea>	BWL	Ziel=Ziel-Quelle	XNZCV	..*******...	
SUB.x <ea>,Dn	BWL	Ziel=Ziel-Quelle	XNZCV	**********	
SUBA.x <ea>,An	WL	Ziel=Ziel-Quelle	.....	**********	
SUBI.x #K,<ea>	BWL	Ziel=Ziel-Quelle	XNZCV	*.*******...	
SUBQ.x #K,<ea>	BWL	Ziel=Ziel-Quelle 0<K<9	XNZCV	********...	
SUBX.x Dn,Dn	BWL	Ziel=Ziel-Quelle-X	XNZCV		
SUBX.x -(An),-(An)	BWL	Ziel=Ziel-Quelle-X	XNZCV		
SWAP Dn		Bit 31-16 <-> Bit 15-0	NZCV		
TAS <ea>		<ea> abgefragt » CCR	NZCV	*.*******...	
		Bit 7 von <ea> = 1			
TRAP #Vektor		PC,SR»Stack,PC=(Vektor)			
TRAPV		TRAP wenn V=1			
TST <ea>	BWL	<ea> abgefragt » Flags	NZCV	*.*******...	
UNLK An		SP=An,An vom Stack			

Die zusätzlichen Befehle des MC 68010

Abkürzungen :

CTRLReg Control-Register (DFC,SFC,VBR,USP)

Befehl		Operation	Flags	000000000011
				012345678901
*MOVEC CTRLReg,Rn	[L]	Rn=CTRLReg		
*MOVEC Rn,CTRLReg	[L]	CTRLReg=Rn		
*MOVES Rn,<ea>	BWL	Ziel=Quelle		..*******...
*MOVES <ea>,Rn	BWL	Ziel=Quelle		..*******...
MOVE CCR,<ea>	[B]	<ea>=CCR		*.*******...
RTD #D		PC vom Stack, SP=SP+D		

Der beim MC 68000 nicht privilegierte Befehl **MOVE SR,<ea>** darf ab dem MC 68010 nur noch im Supervisor-Mode benutzt werden !

Anhang B

Die Assemblerdirektiven

BSS
Leitet eine BSS-Sektion ein. Memory-Typ ist MEMF_PUBLIC.

SECTION <Name>,<Typ>,<Memory-Typ>
Leitet eine neue Sektion **<Typ>** ein.

<Name>	:	Name der Sektion
<Typ>	:	Sektionstyp CODE(TEXT), DATA oder BSS
<Memory-Typ>	:	FAST,CHIP oder PUBLIC

NOMEXP
Verbietet die Macroexpansion im Listing.

CEND
Erzeugt einen END-Befehl für eine Copperliste (= **CWAIT -1,-2**).

DATA
Leitet eine DATA-Sektion ein. Memory-Typ ist **MEMF_PUBLIC**.

NOLIST
Unterdrückt die Ausgabe in ein Listing.

ALIGN.x
Bringt die Adresse auf Wortgrenze (**.W**) bzw. auf Langwortgrenze (**.L**).

EQU
Weist einer Variablen einen Wert zu.

XDEF <Variable>[,<Variable>...]
Macht dem Linker die Variable(n) bekannt.

CODE
Leitet eine CODE-Sektion ein. Memory-Typ ist **MEMF_PUBLIC**.

PUBLIC <Variable>[,<Variable>...]
Eine Alternative zu **XDEF**.

DB <Konstante>[,<Konstante>...]
Assembliert Byte-Konstante(n). (= **DC.B**)

IFcc <Term1>[,<Term2>]
Die Assemblierung wird fortgesetzt, wenn **cc=TRUE**. Ist **cc=False**, wird erst nach einem **ELSE** bzw. dem zugehörigen **ENDIF** weiterassembliert.

ENDC
Beendet eine IF-Konstruktion (= **ENDIF**).

DC.x <Konstante>[,<Konstante>...]
Assembliert Konstante(n) der Größe x (**B/W/L**).

PROGRAM <Name>
Gibt der Programmunit den Namen <Name> (nur bei Link-Files).

DCB.x <Größe>,<Füllwert>
Assembliert einen Block aus **<Größe>** Konstanten der Länge x (**B/W/L**) und initialisiert den Block mit **<Füllwert>**.

IDENTIFY <Name>
Eine Alternative zu PROGRAM.

ENDIF
Beendet eine **IF**-Konstruktion.

CLMOVE #<Wert>,<Register>
Erzeugt zwei **CMOVE**-Befehle (Copper). Ein Langwort **<Wert>** wird in **<Register>** und **<Register>+2** übertragen.

ENDM
Schließt eine Macrodefinition ab.

TEXT
Eine Alternative zu **CODE**.

CNOP <Align>,<Dist>
Macht die Adresse durch **<Align>** (2 oder 4) teilbar und addiert **<Dist>**.

CWAIT <YPos>,<XPos>[{YMask,XMask}BFD]
Erzeugt einen **CWAIT**-Befehl (Copper), der auf **YPos [AND YMask]** und **XPos [AND XMask]** und eventuell auf den Blitter (**BFD=0**) wartet.

CMOVE #<Wert>,<Register>
Erzeugt einen **CMOVE**-Befehl (Copper), der den angegebenen Wert in eine Register der Custom-Chips überträgt.

ELSE
Kehrt das Ergebnis einer **IF**-Abfrage um. War das Ergebnis **FALSE**, werden jetzt die auf **ELSE** folgenden Befehle assembliert.

ELSEIF
Eine Alternative zu **ELSE**.

CSKIP <YPos>,<XPos>[{YMask,XMask}BFD]
Erzeugt einen **CSKIP**-Befehl (Copper).

MACRO
Leitet eine Macrodefinition ein.

RSRESET
Setzt den internen Zähler __RS auf NULL.

PSTRING <String>[,<String>]
Der (Die) String(s) werden mit einem führenden Längenbyte assembliert.

PSTR
Eine Alternative zu **PSTRING**.

RSSET <Wert>
Initialisiert den internen Zähler __RS mit **<Wert>**.

INCBIN <Datei>[,<Länge>]
Eine Datei wird in den Programmcode integriert. **<Länge>** gibt an, wieviele Bytes geladen werden sollen.

GLOBAL <Variable>[,<Variable>]
Eine Alternative zu **XDEF**.

INCDIR <Pfad>
Meldet den Pfad als Suchpfad für **INCLUDE**-Dateien an.

DL <Konstante>[,<Konstante>...]
Assembliert Langwort-Konstante(n) (= **DC.L**).

INCLUDE <Datei>
Leitet die Assemblierung in die angegebene Datei um.

SET
Weist einer Variablen einen Wert zu.

ODDERR
Words und Longwords an ungeraden Adressen führen zu einer Fehlermeldung.

BLK.x <Größe>[,<Füllwert>]
Eine Alternative zu **DCB**.

IBYTES <Datei>[,<Länge>]
Eine Alternative zu **INCBIN**.

EQUATE
Eine Alternative zu **EQU**.

EVEN
Macht die Adresse wieder durch zwei teilbar.

XREF <Variable>[,<Variable>]
Die Variable(n) werden importiert (nur in Link-Files).

ENTRY <Variable>[,<Variable>]
Eine Alternative zu **XDEF**.

RS.x <Anzahl>
Addiert zum RS-Zähler **<Anzahl>** Bytes (**B**), Words (**W**) oder Longwords (**L**).

Steht vor **RS.x** eine Variable, wird ihr der alte Wert von **__RS** zugewiesen.

CSTRING <String>[,<String>]
Die Strings werden mit einem abschließenden **NULL**-Byte assembliert.

CSTR <String>[,<String>]
Eine Alternative zu **CSTRING**.

ODDOK
Words und Longwords an ungeraden Adressen sind erlaubt !

OPT
Hat keine Wirkung (aus Kompatibilitätsgründen vorhanden).

IFC <String1>,<String2>
Vergleicht zwei Strings und gibt **TRUE** wenn beide Strings gleich sind und auch die gleiche Länge haben.

END
Beendet die Assemblierung.

IFD <Variable>
Stellt fest, ob die Variable definiert ist und gibt **TRUE** wenn ja.

DS.x <Größe>[,<Füllwert>]
Eine Alternative zu **DCB**.

MEXIT
Beendet vorzeitig die Bearbeitung eines Macros.

EXTERN <Variable>[,<Variable>]
Eine Alternative zu **XREF**.

DL <Konstante>[,<Konstante>...]
Assembliert Wort-Konstante(n) (= **DC.W**).

MEXP
Erlaubt die Macroexpansion im Listing.

LIST
Gibt die Ausgabe in ein Listing wieder frei.

BASEREG

BASEREG OFF verbietet die Optimierung absoluter Adressen relativ zu einem Adressregister. **BASEREG An,<Basis>** gibt die Optimierung relativ zu An mit der **<Basis>** frei.

FAIL <Meldung>
Erzeugt eine Benutzerfehlermeldung

Anhang C: Die Fehlermeldungen

DOS-Fehler beim Dateihandling

032 DOS-Fehler xxx beim LOCK auf Verzeichnis <Name>
033 DOS-Fehler xxx beim Laden der Quelldatei <Name>
034 DOS-Fehler xxx beim Öffnen der Objektdatei <Name>
035 DOS-Fehler xxx beim Schreiben der Objektdatei <Name>
036 DOS-Fehler xxx beim Öffnen der Includedatei <Name>
037 DOS-Fehler xxx beim Öffnen der Binärdatei <Name>
038 DOS-Fehler xxx beim Lesen der Binärdatei <Name>
039 DOS-Fehler xxx beim Schreiben der Prelistingdatei <Name>
040 DOS-Fehler xxx beim Öffnen der Prelistingdatei <Name>
041 DOS-Fehler xxx beim Öffnen Listingdatei <Name>
042 DOS-Fehler xxx beim Lesen der Prelistingdatei <Name>
043 DOS-Fehler xxx beim Schreiben der Listingdatei <Name>
044 DOS-Fehler xxx beim SEEK in der Prelistingdatei <Name>
046 DOS-Fehler xxx beim Öffnen der Fehlerdatei <Name>
047 DOS-Fehler xxx beim Öffnen der Symboltabelle <Name>

Fehler in der Sektionierung

048 Programm enthält zu viele Sektionen (maximal 256)
049 Falscher Sektionstyp (nur CODE,DATA,BSS)
050 Gleichnamige Sektion eines anderen Types existiert bereits
051 Falscher Speichertyp (nur PUBLIC,CHIP,FAST)
052 Gleichnamige Sektion mit anderen Speichertypen existiert bereits
053 Codeproduzierende Direktiven im BSS-Hunk nicht zulässig
054 Vor „ORG" dürfen keine codeproduzierenden Direktiven stehen
055 Basisadresse nach „ORG" muß gerade sein

MACRO-Fehler

056 Falsche Parameterbezeichnung (nur \0...\9 und \A...\Z)
057 Name vor MACRO fehlt
058 Reservierter Name
059 Makroname muß mit einem Buchstaben oder „_" beginnen
060 Makroname kollidiert mit Label
061 Makroname kollidiert mit absolutem Symbol
062 Makroname kollidiert mit importiertem Symbol
063 Makroname kollidiert mit Registersymbol

064 Illegale Redefinition eines Makros
065 Makroname kollidiert mit Registerlistensymbol
066 MACRO darf nicht innerhalb eines Makros vorkommen
067 MACRO ohne zugehöriges ENDM innerhalb der Includedatei
068 MACRO ohne zugehöriges ENDM
069 Vor ENDM darf kein Label stehen
070 Unzulässiges leeres Makro
071 Zu viele Makroparameter (maximal 36)
072 END darf nicht innerhalb eines Makros vorkommen
073 Falsche IF/ENDIF-Paare im Makro
074 MEXIT darf nur innerhalb eines Makros vorkommen
075 Nach Schlüsselwort muß unmittelbar „(„ folgen
076 Makrofunktion nicht implementiert
077 Makrofonktionen zu tief geschachtelt (maximal 16fach)
078 Argument einer Makrofunktion muß mit „)“ enden
079 VALOF() darf nur innerhalb eines Makros vorkommen
080 STRLEN() darf nur innerhalb eines Makros vorkommen
081 LEFT() darf nur innerhalb eines Makros vorkommen
082 RIGHT() darf nur innerhalb eines Makros vorkommen
083 MID() darf nur innerhalb eines Makros vorkommen
084 UPPER() darf nur innerhalb eines Makros vorkommen
085 *VALOF() : Ausdruck enthält undefinierte Symbole
086 *VALOF() : Argument muß ein Symbol sein
087 *STRLEN() : Fehlende Argumente
088 *LEFT() : Erster Operand fehlt
089 *LEFT() : Zweiter Operand fehlt
090 *LEFT() : Zweiter Operand enthält undefinierte Symbole
091 *LEFT() : Zweiter Operand muß ein absoluter Ausdruck sein
092 *LEFT() : Zweiter Operand muß größer als NULL sein
093 *RIGHT() : Erster Operand fehlt
094 *RIGHT() : Zweiter Operand fehlt
095 *RIGHT() : Zweiter Operand enthält undefinierte Symbole
096 *RIGHT() : Zweiter Operand muß ein absoluter Ausdruck sein
097 *RIGHT() : Zweiter Operand muß größer als NULL sein
098 *MID() : Erster Operand fehlt
099 *MID() : Zweiter Operand fehlt
100 *MID() : Zweiter Operand enthält undefinierte Symbole
101 *MID() : Zweiter Operand muß ein absoluter Ausdruck sein
102 *MID() : Zweiter Operand muß größer als NULL sein
103 *MID() : Dritter Operand fehlt
104 *MID() : Dritter Operand enthält undefinierte Symbole
105 *MID() : Dritter Operand muß ein absoluter Ausdruck sein

106 *MID() : Startposition befindet sich nicht mehr im String
107 *MID() : Dritter Operand muß größer als NULL sein
108 *UPPER() : Fehlende Argumente

Fehler beim Parsen einer Zeile

112 Unzulässige Zeichen im Labelfeld
113 Lokales Label muß mindestens ein Zeichen lang sein
114 Label darf keine Längenangabe haben
115 Lokales Label muß mit „$" enden
116 Unzulässige Zeichen im Opcodefeld
117 Unzulässige Zeichen am Zeilenende
118 Unzulässige Zeichen im Operandenfeld

Fehler bei der Labeldefinition

120 Illegale Redefinition eines Labels
121 Label kollidiert mit absolutem Symbol
122 Label kollidiert mit importiertem Symbol
123 Label kollidiert mit Registersymbol
124 Label kollidiert mit Makro
125 Label kollidiert mit Registerlistensymbol

Fehler im math. Ausdruck

128 Stacküberlauf - Ausdruck zu komplex
129 Division durch NULL
130 Modulo NULL
131 Addition zweier Label nicht zulässig
132 Verschieben nur um 1...32 Bit(s) möglich
133 Label darf nicht von absolutem Ausdruck subtrahiert werden
134 Importiertes Symbol darf nicht von absolutem Ausdruck subtrahiert werden
135 Operation mit Label nicht zulässig
136 Label aus verschiedenen Sektionen dürfen nicht voneinander subtrahiert werden
137 Klammerfehler
138 Operationszeichen muß „<<" heißen
139 Operationszeichen muß „>>" heißen
140 Unbekanntes Zahlenformat
141 Hexadezimalzahl darf nur 0...9 und A...F enthalten
142 Binärzahl darf nur 0 und 1 enthalten

143 Oktalzahl darf nur 0...7 enthalten
144 ASCII-Zahl darf maximal 4stellig sein
145 Illegale Vorwärtsreferenz
146 Symbol nicht definiert
147 Hexadezimalzahl überschreitet den 32-Bit-Bereich
148 Binärzahl überschreitet den 32-Bit-Bereich
149 Dezimalzahl überschreitet den 32-Bit-Bereich
150 Oktalzahl überschreitet den 32-Bit-Bereich
151 Makro im Ausdruck nicht zulässig
152 Registersymbol im Ausdruck nicht zulässig

Fehler im Addressmode

160 Befehl oder Addressmode erst ab MC68010 erlaubt
161 Befehl oder Addressmode erst ab MC68020 erlaubt
162 Befehl nur für FPU MC 68881/2 zulässig
163 Unmittelbare Konstante vom falschen Typ
164 Base-Displacement muß absolut,relativ oder importiert sein
165 Outer-Displacement muß absolut,relativ oder importiert sein
166 Addressmode enthält Syntaxfehler
167 Unzulässiges Basisregister (nur An oder PC)
168 Unzulässiges Indexregister (nur An oder Dn)
169 „PC“ als Indexregister nicht zulässig
170 Längenangabe muß .W oder .L sein
171 Scalar muß ein absoluter Ausdruck sein
172 Scalar darf nicht NULL sein
173 Scalar darf nur 1,2,4 oder 8 sein
174 16-Bit-Distanz muß absolut oder importiert sein
175 Distanz überschreitet vorzeichenbehafteten 16-Bit-Bereich
176 8-Bit-Distanz muß absolut oder importiert sein
177 Distanz überschreitet vorzeichenbehafteten 8-Bit-Bereich
178 PC-relativ adressiertes Ziel ist weiter als +/-32768 Bytes entfernt
179 PC-relativ adressiertes Ziel ist weiter als +/-128 Bytes entfernt
180 Komma erwartet
181 „)“ erwartet
182 „:“ erwartet
183 „{„ erwartet
184 „}“ erwartet
185 Bitfeld-Offset muß ein absoluter Ausdruck sein
186 Bitfeld-Offset muß 0...31 sein
187 Bitfeld-Offset muß „Dn“ sein
188 Bitfeld-Width muß ein absoluter Ausdruck sein

189 Bitfeld-Width muß 0...31 sein
190 Bitfeld-Width muß „Dn“ sein
191 Byteoperation mit Adreßregister nicht zulässig
192 Operandenlänge .L nicht zulässig
193 Operandenlänge .B nicht zulässig
194 Operandenlänge muß .B sein (kann weggelassen werden)
195 Operandenlänge muß .W sein (kann weggelassen werden)
196 Operandenlänge muß .L sein (kann weggelassen werden)
197 Addressmode für diesen Befehl nicht zulässig
198 Spezialregister <Reg> nicht zulässig
199 Registerliste nicht erlaubt
200 Registerliste erwartet
201 Register erwartet
202 Addressmode „Dn“ nicht erlaubt
203 Addressmode „An“ nicht erlaubt
204 Addressmode „(An)“ nicht erlaubt
205 Addressmode „(An)+“ nicht erlaubt
206 Addressmode „-(An)“ nicht erlaubt
207 Addressmode „(bd,An)“ nicht erlaubt
208 Addressmode „(bd,An,Rn)“ nicht erlaubt
209 Addressmode „($xxx).w“ nicht erlaubt
210 Addressmode „($xxx).l“ nicht erlaubt
211 Addressmode „(bd,PC)“ nicht erlaubt
212 Addressmode „(bd,PC,Rn)“ nicht erlaubt
213 Addressmode „#k“ nicht erlaubt
214 PC-relativ adressiertes Ziel liegt in einer anderen Sektion
215 FAIL <Zeile>
216 Blockgröße darf nicht negativ sein
217 Befehl oder Addressmode nur für PMMU 68030/68851 erlaubt
218 Befehl oder Addressmode nur für PMMU 68030 erlaubt
219 Befehl oder Addressmode nur für PMMU 68851 erlaubt
220 Befehl oder Addressmode nur für Copper erlaubt

Fehler im ersten Befehlsoperanden

224 Erster Operand muß „Dn“ sein
225 Erster Operand muß „Dn“ oder „An“ sein
226 Erster Operand muß „Dn“ oder „-(An)“ sein
227 Erster Operand muß „Dn“ oder „(d16,An)“ sein
228 Erster Operand muß „Dn“ oder „#k“ sein
229 Erster Operand muß „Dn:Dn“ sein
230 Erster Operand muß „An“ sein

231 Erster Operand muß „-(An)“ sein
232 Erster Operand muß „#k“ sein
233 Erster Operand muß absolut sein
234 Erster Operand muß 0...7 sein
235 Erster Operand muß 1...8 sein
236 Erster Operand muß 0...15 sein

Fehler im zweiten Befehls-Operanden

240 Zweiter Operand muß „Dn“ sein
241 Zweiter Operand muß „Dn“ oder „An“ sein
242 Zweiter Operand muß „Dn“ oder „-(An)“ sein
243 Zweiter Operand muß „Dn:Dn“ sein
244 Zweiter Operand muß „An“ sein
245 Zweiter Operand muß „(An)+“ sein
246 Zweiter Operand muß „-(An)“ sein
247 Zweiter Operand muß „(d16,An)“ sein
248 Zweiter Operand muß „#k“ sein
249 Zweiter Operand muß „CCR“ oder „SR“ sein
250 Zweiter Operand muß Spezialregister sein
251 Zweiter Operand überschreitet den 8-Bit-Bereich
252 Zweiter Operand muß absolut sein
253 Zweiter Operand muß 0...15 sein

Fehler im dritten Befehls-Operanden

256 Dritter Operand muß „(Rn):(Rn)“ sein
257 Dritter Operand muß „#k“ sein
258 Dritter Operand muß absolut sein
259 Dritter Operand muß 0...7 sein

Fehler in den IF-Konstruktionen

264 Falsche IF/ENDIF-Paare
265 Falsche IF/ENDIF-Paare in Include-Datei
266 ENDIF ohne zugehöriges IF
267 ELSE ohne zugehöriges IF/ENDIF-Paar
268 Mehrfaches ELSE zwischen einem IF/ENDIF-Paar
269 Argument von IFD/IFND muß ein Symbol sein
270 Erstes Argument von IFC/IFNC fehlt
271 Zweites Argument von IFC/IFNC fehlt

272 Ausdruck nach IFcc enthält undefinierte Symbole
273 Ausdruck nach IFcc muß absolut sein

Spezielle Fehler bei Branch-Befehlen

280 Branch-Ziel muß ein Label sein
281 Branch-Ziel muß in der gleichen Sektion liegen
282 Branch-Ziel weiter als +/-32768 Bytes entfernt
283 Branch-Ziel weiter als +/-128 Bytes entfernt

Spezielle Fehler bei EQU

288 Symbolname vor EQU fehlt
289 Symbol kollidiert mit Label
290 Illegale Redefinition eines Symboles
291 Symbol kollidiert mit importiertem Symbol
292 Symbol kollidiert mit Registersymbol
293 Symbol kollidiert mit Makro
294 Symbol kollidiert mit Registerlistensymbol

Spezielle Fehler bei EQUR

296 Symbolname vor EQUR fehlt
297 Registersymbol kollidiert mit Label
298 Registersymbol kollidiert mit absolutem Symbol
299 Registersymbol kollidiert mit importiertem Symbol
300 Registersymbol kollidiert mit Makro
301 Registersymbol kollifiert mit Registerlistensymbol
302 Ausdruck nach EQUR muß ein Register sein

Spezielle Fehler bei REG

304 Symbolname vor REG fehlt
305 Registerlistensymbol kollidiert mit Label
306 Registerlistensymbol kollidiert mit absolutem Symbol
307 Registerlistensymbol kollidiert mit importiertem Symbol
308 Registerlistensymbol kollidiert mit Registersymbol
309 Registerlistensymbol kollidiert mit Makro

Spezielle Fehler bei SET

312 Symbolname vor SET fehlt
313 Ausdruck nach SET enthält undefinierte Symbole

Spezielle Fehler bei CNOP/ALIGN

320 Operand von CNOP/ALIGN enthält undefinierte Symbole
321 Alignment muß 1...4096 sein

Spezielle Fehler bei XDEF

328 Argument von XDEF muß ein Symbol sein
329 Symbol ist bereits exportiert
330 Ein importiertes Symbol darf nicht exportiert werden
331 Ein Registersymbol kann nicht exportiert werden
332 Ein Makro kann nicht exportiert werden
333 Ein Registerlistensymbol kann nicht exportiert werden

Spezielle Fehler bei XREF

336 Nur globale Symbole dürfen exportiert werden
337 Importiertes Symbol kollidiert mit Label
338 Importiertes Symbol kollidiert mit absolutem Symbol
339 Symbol ist bereits importiert
340 Importiertes Symbol kollidiert mit Registersymbol
341 Importiertes Symbol kollidiert mit Makro
342 Importiertes Symbol kollidiert mit Registerlistensymbol
343 XREF/XDEF darf nur in zu linkenden Programmen vorkommen

Spezielle Fehler bei DC.x

352 Bytekonstante muß absolut oder importiert sein
353 Wortkonstante muß absolut oder importiert sein
354 Langwortkonstante muß relativ,absolut oder importiert sein
355 Bytekonstante muß -128...255 sein
356 Wortkonstante muß -32768...65535 sein

Spezielle Fehler bei RS.x

360 Symbolname vor RS.x fehlt
361 Ausdruck nach RS.x enthält undefinierte Symbole

362 Ausdruck nach RS.x muß absolut sein
363 Ausdruck nach RSSET enthält undefinierte Symbole
364 Ausdruck nach RSSET muß absolut sein

Spezielle Fehler bei INCBIN

368 INCBIN : Längenangabe enthält undefinierte Symbole
369 INCBIN : Längenangabe muß ein absoluter Ausdruck sein
370 INCBIN : Längenangabe muß größer als NULL sein

Spezielle Fehler bei BASEREG

376 Unbekannte BASEREG-Option
377 Basisadresse muß ein Label sein

Allgemeine Fehler

384 Fataler Fehler - Kein Speicher mehr
385 Unzulässige Angabe der Operandenlänge
386 Unzulässige Operanden
387 Fehlende Operanden
388 Direktive <%s> nicht implementiert
389 String endet nicht mit „<Hochkomma>“
390 Dateiname nach INCLUDE fehlt
391 Code-Phase-Fehler : Letzter Pass xxx Bytes, jetziger Pass xxx Bytes
392 PC seit dem letzten Pass verändert
393 Assembler abgebrochen ...

Interne Assembler-Fehler

400 #01
401 #02

Diese Fehler weisen auf einen internen Assemblerfehler hin. Bitte melden Sie diesen Fehler und eine genaue Beschreibung des Programmes, bei dem er auftrat, damit er möglichst schnell behoben werden kann.

Fehler bei FPU-Operationen

Fehler im mathematischen Ausdruck
408 Fließkommazahlenüberlauf
409 Exponent für Single-Zahl zu groß
410 Exponent erwartet
411 Fließkommazahl erwartet
412 FPU-Format Packed decimal nur mit FPU verfügbar
413 mathieeedoubbas- und mathieeedoubtrans.library fehlen

Fehler im Addressmode

416 Operandenlänge nur für FPU-Befehle zulässig
417 FPU-Register im Operanden nicht zulässig
418 Einregistersyntax für diesen Befehl nicht zulässig
419 Operandenlänge muß .X sein (kann weggelassen werden)
420 Quelloperand Dn nur bei Single-Datentypen (.B,.W,.L,.S) zulässig
421 Statischer K-Faktor {#k} muß ein absoluter Ausdruck sein
422 Statischer K-Faktor {#k} muß -64...63 sein
423 K-Faktor muß „{Dn}“ oder „{#k}“ sein
424 Kontrollregister muß „FPCR“,“FPSR“ oder „FPIAR“ sein
425 Addressmode „An“ nur für FPIAR zulässig
426 ROM-Offset #k muß ein absoluter Ausdruck sein
427 Unzulässiger ROM-Offset - Siehe MC 68881 Dokumentation
428 Operandenlänge muß .L oder .X heißen
429 Registerliste darf nur FPU-Kontrollregister enthalten
430 Registerliste darf nur FPU-Datenregister enthalten
431 Registerliste darf nur aus einem Register bestehen
432 FPn nach „:“ erwartet
Fehler im ersten Operanden
448 Erster Operand muß „FPn“ sein

Fehler im zweiten Befehlsoperanden

464 Zweiter Operand muß „FPn“ sein

Fehler im Addressmode

472 MMU-Register im Operanden nicht zulässig
473 Addressmode „Dn“ oder „An“ für DRP,SRP oder CRP nicht zulässig
474 PCSR darf nur gelesen werden
475 MMUSR bei PMOVEFD unzulässig

476 Erster Operand bei FMOVEFD darf kein Kontrollregister sein
477 Vierter Operand muß An sein

Fehler im ersten Operanden

488 Erster Operand muß „#k", „Dn", „SFC" oder „DFC" sein
489 Erster Operand muß „An" oder „VAL" sein

Fehler im zweiten Operanden

496 Zweiter Operand muß MMU-Kontrollregister sein

Fehler bei Copper-Operationen

504 Copper-Zielregisteroffset ungerade
505 Y-Position muß 0...255 sein
506 X-Position muß 0...511 sein
507 Y-Maske muß ein absoluter Ausdruck sein
508 X-Maske muß ein absoluter Ausdruck sein

Anhang D: Register

Name	68000	68010	68020	68030	68881	68851
CRP						*
SRP						*
DRP						*
TC						*
PCSR						*
PSR/MMUSR				*		*
CAL						*
VAL						*
SCC						*
AC						*
BAC0 ... BAC7						*
BAD0 ... BAD7						*
TT0 ... TT1				*		
FPIAR					*	
FPSR					*	
FPCR					*	
FP0 ... FP7					*	
ISP			*	*		
MSP			*	*		
VBR		*	*	*		
SFC		*	*	*		
DFC		*	*	*		
CACR			*	*		
CAAR			*	*		
USP	*	*	*	*		
CCR	*	*	*	*		
SR	*	*	*	*		
PC	*	*	*	*		
SP	*	*	*	*		
A0 ... A7	*	*	*	*		
D0 ... D7	*	*	*	*		

Anhang E, Befehle

Name	68000	68010	68020	68030	68881	68851	Copper
ABCD	*	*	*	*			
ADD	*	*	*	*			
ADDA	*	*	*	*			
ADDI	*	*	*	*			
ADDQ	*	*	*	*			
ADDX	*	*	*	*			
AND	*	*	*	*			
ANDI	*	*	*	*			
ASL	*	*	*	*			
ASR	*	*	*	*			
Bcc	*	*	*	*			
BCHG	*	*	*	*			
BCLR	*	*	*	*			
BFCHG			*	*			
BFCLR			*	*			
BFEXTS			*	*			
BFEXTU			*	*			
BFFFO			*	*			
BFINS			*	*			
BFSET			*	*			
BFTST			*	*			
BKPT			*	*			
BSET	*	*	*	*			
BTST	*	*	*	*			
CALLM			*				
CAS			*	*			
CAS2			*	*			
CHK	*	*	*	*			
CHK2			*	*			
CLR	*	*	*	*			
CMP	*	*	*	*			
CMP2			*	*			
CMPA	*	*	*	*			
CMPI	*	*	*	*			
CMPM	*	*	*	*			
CEND							*

Name	68000	68010	68020	68030	68881	68851	Copper
CLMOVE							*
CMOVE							*
CSKIP							*
CWAIT							*
DBcc	*	*	*	*			
DIVS	*	*	*	*			
DIVS.L			*	*			
DIVSL			*	*			
DIVU	*	*	*	*			
DIVU.L			*	*			
DIVUL			*	*			
EOR	*	*	*	*			
EORI	*	*	*	*			
EXG	*	*	*	*			
EXT	*	*	*	*			
EXTB			*	*			
FABS					*		
FACOS					*		
FADD					*		
FASIN					*		
FATAN					*		
FATANH					*		
FBcc					*		
FCMP					*		
FCOS					*		
FCOSH					*		
FDBcc					*		
FDIV					*		
FETOX					*		
FETOXM1					*		
FGETEXP					*		
FGETMAN					*		
FINT					*		
FINTRZ					*		
FLOG10					*		
FLOG2					*		
FLOGN					*		
FLOGNP1					*		
FMOD					*		
FMOVE					*		

Name	68000	68010	68020	68030	68881	68851	Copper
FMOVECR					*		
FMOVEM					*		
FMUL					*		
FNEG					*		
FNOP					*		
FREM					*		
FRESTORE					*		
FSAVE					*		
FSCALE					*		
FScc					*		
FSGLDIV					*		
FSGLMUL					*		
FSIN					*		
FSINCOS					*		
FSINH					*		
FSQRT					*		
FSUB					*		
FTAN					*		
FTANH					*		
FTENTOX					*		
FTRAPcc					*		
FTST					*		
FTWOTOX					*		
ILLEGAL	*	*	*	*			
JMP	*	*	*	*			
JSR	*	*	*	*			
LEA	*	*	*	*			
LINK	*	*	*	*			
LSL	*	*	*	*			
LSR	*	*	*	*			
MOVE	*	*	*	*			
MOVEA	*	*	*	*			
MOVEC		*	*	*			
MOVEM	*	*	*	*			
MOVEP	*	*	*	*			
MOVES		*	*	*			
MULS	*	*	*	*			
MULS.L			*	*			
MULU	*	*	*	*			
MULU.L			*	*			

Name	68000	68010	68020	68030	68881	68851	Copper
NBCD	*	*	*	*			
NEG	*	*	*	*			
NEGX	*	*	*	*			
NOP	*	*	*	*			
NOT	*	*	*	*			
OR	*	*	*	*			
ORI	*	*	*	*			
PACK			*	*			
PBcc						*	
PDBcc						*	
PFLUSH				*		*	
PFLUSHA				*		*	
PFLUSHS						*	
PFLUSHR						*	
PLOADR				*		*	
PLOADW				*		*	
PMOVE				*		*	
PMOVEFD				*		*	
PRESTORE						*	
PSAVE						*	
PScc						*	
PTESTR				*		*	
PTESTW				*		*	
PTRAPcc						*	
PVALID						*	
RESET	*	*	*	*			
ROL	*	*	*	*			
ROR	*	*	*	*			
ROXL	*	*	*	*			
ROXR	*	*	*	*			
RTD		*	*	*			
RTE	*	*	*	*			
RTM			*				
RTR	*	*	*	*			
RTS	*	*	*	*			
SBCD	*	*	*	*			
Scc	*	*	*	*			
STOP	*	*	*	*			
SUB	*	*	*	*			
SUBA	*	*	*	*			

Name	68000	68010	68020	68030	68881	68851	Copper
SUBI	*	*	*	*			
SUBQ	*	*	*	*			
SUBX	*	*	*	*			
SWAP	*	*	*	*			
TAS	*	*	*	*			
TRAP	*	*	*	*			
TRAPcc			*	*			
TST	*	*	*	*			
UNLK	*	*	*	*			
UNPK			*	*			

Index

! 220
!= 220
» ..... 101, 22, , 437, 456
«= 223, 226
#endif 164, 471
#error 164
#if 164, 471
#include 163, 259
#line 163
#pragma 259, 473
#pragma amicall 182, 301, 476
#pragma header 157, 185
% 226
%= 223, 226
& 221, 226
&& 220, 226
&= 223, 226
* 453
*= 223, 228
+ 218, 437
+= 223, 228
, ... 224
- 218, 453
-= 223
-g20 158
-g30 158
3.0-Standard 509
< 220
<< 218
<= 220
= 223, 22, 256, 441
== 220
?: 222
^ 221, 226
^= 223, 226
_MOVEMBYTES 739
_wbmain 158
__COMPMODE__ 481
__cplusplus 481
__DATE2__ 157
__DATE__ 480
__FILE__ 480
__LINE__ 480
__MAXON__ 157, 481
__MEMFLAGS 604
__RS 739, 754
__STDC__ 481
__TIME__ 480
| 101, 22, 226

|= 223, 226
|| 221, 226
~ 174, 221

A

Abgeleitete Klassen 366
Abhängigkeiten 49
Abort 534
Abrufen der Projektübersicht 692
Abs 529
Absoluter Code 750
Abspielen 105
Abstrakte Klassen 381
Acos 564
Addition 218
Adressierungsarten 735
ALIGNment 747
Amiga-Betriebssystem 300
AmigaGuide 681
AmigaGuide-Projekte 716
Anhalten 105
ANSI C++ 509
ANSI-Modus 258
Antiquitäten 337

Anweisungen 230
- break 240
- case 242
- continue 240
- do 237
- for 236
- if 230
- return 244
- switch 242
- Throw 512
- while 235

Anwendungsgebiete für Onlineprojekte 700
AppIcon 659
Arcsin 512
ARexx 682, 701

ARexx-Befehle 39
- addtx 40
- addtxobj 40
- break 39
- compile 39
- editscript 41
- errors 40
- errorscript 41
- exefile 40
- getinfo 41

info 40
leavemk 40
loadmk 40
newmk 40
objectfile 40
opcomp 40
openmake 40
oplink 40
opsys 40
Options results: 41
quit 39
reopen 39
run 39
saveconf 40
savesession 40
setopt 40
shutdown 39
ARexx-Port 123, 151
Argc 285
Argumemt-Matching 501
Argumente 62, 657
Argumentliste 246, 493
Argv 285
Asctime 586
Asin 564

Assembler 735
Optionen 735
Variablen 738

Assert 556
Atan 564
Atan2 565
Atexit 475, 534
Atof 529
Atoi 528
Atol 528
Auflösung wählen 113
Aufnehmen 105
Aufteilung des Textes 721
Aufzählungen 316
Ausnahme-Deklarationen 521
Ausnahmebehandlung 623
Ausnahmen 512
Ausschneiden 93
Auswertungsregeln 226
Auszeichnen 102
Auto 171
Auto-Indent 103
Auto-Save 114
Automatische Daten 292
Autoren V-2

B

BACKSPACE 87
Basisklassen 396
Bedingungen 230
Benutzerdefinierte Menüs 117
Benutzung der Onlineprojekte 699
Bezeichner 209
Bibliotheken 159
Bit-Manipulationen 219
Bitfelder 332
Blättern im Projekt 691
Block drucken 95
Block einlesen 95

Blockoperationen
- Cut, Copy, Paste 92
- Einrücken 95
- Laden, Speichern, Drucken 94
- Spaltenblöcke 94
- Umwandeln 95

Bookmarks 102
BOOPSI Gadgets 630
Break 240, 474
Breakpoint 664
Breakpoints 664
Bsearch 536
BUFSIZ 551

C

C-Modus 258
Calloc 532
Case 180, 242
Casting 228
Catch 28, 158, 512, 519
Catch-Handler 515
Ceil 565
Char 211
CHAR_BIT 561
CHAR_MAX 561
Class Exception 605
CleanupModules 65, 615
Clearerr 549
CLI 86, 285

CLI-Compiler 69
- Ausgaben 72
- Codeerzeugung 75
- Linker 79
- Optionen 71

Clock 584
CLOCKS_PER_SEC 584
Clock_t 582
Codeerzeugung 158
Compiler is in Panic! 186

Compiler-Fehlermeldungen 99
"##" at end of Macro 162
"#pragma header" may appear on file scope only 185
"(" expected 179
")" expected 162, 166
"," expected 163
"." need a class expression as left operand 174
":" expected 167, 180
" expected 165
"amicall": int/pointer variable expected 182
"amicall": non-overloaded function expected 182
"break" without matching iteration or switch statement 179
"case" requires constant expression 180
"case" without "switch" 180
"catch" expected 184
"continue" used outside iteration statement 179
"default" without "switch" 180
"else" without "if" 179
"Garbage" at end of preprocessor command 163
"id1" defined in classes "id2" and "id3" 176
"inline" used after definition or use of function 171
"inline" used with storage class specifier 171
"sizeof" needs an expression or a type in "()""s 174
"throw": Expression expected 185
"while" expected 179
"]" expected 169
"{" expected 184
"}" expected 169
+= / -= used on pointer: integral expr. as 2nd operand required 179
A global anonymous union must be declared "static" 165
A template declaration may only appear as a global declaration 183
Access to ambiguous base class 169
Access to private base class 172
Access to variable that was constructed under condition 168
Address operator "&" needs a l-value 170
Allocation of vectors: no initializer allowed 176
Ambiguous conversion function 173
Ambiguous function call 171
Ambiguous: use constructor or user conversion? 175
Anonymous union: Illegal member "xxx" 168
Argument missing 171
Arguments do not match function template 184
Assignment: Left value must be L-Value 172
Assignment: Type conflict 170
At least one "#endif" is missing 164
Attempt to assign to const object 178

Attempt to assign to undefined class 179
Attempt to change access of member "xxx" of class "yyy" 168
Automatic variable used outside of its function 182
Binary operator "op" used on illegal arguments 174
Bitfield member has illegal size 169
Calling pure virtual function 177
Can not copy file "source" to "dest" 163
Can not define qualified identifier 167
Can not mix declarations and other statements in C mode 181
Can not open file "name" 163
Can not open header file "name" 185
Can not open math library 164
Can not overload old-style functions 180
Can not overload operator "delete" 176
Can not subtract pointers of different types 178
Can not take address of a "register" object 182
Can not take address of register parameter 181
Can't convert "qualified type*" to plain "type*" 172
Can't reference "const TYPE" with plain "TYPE &" 182
Can't convert virtual base class to derived class 177
Char constant: unmatched 162
Class "id1" is abstract 177
Class members can not get initialized 167
Class name collides with template name 184
Compiler is in Panic! 163, 186
Compiler stack overflow! 164
Condition expected 178
Conditional expression: ":" expected 178
Constant appears twice in same "switch" statement 180
Constant expression as template argument expected 183
Constant integer expression expected 169
Constant String is too long 169
Constructor initializer: "{" expected 177
Conversion "int" - "enum": allowed in C mode only 172
Conversion function not found 173
Conversion functions may not be static 173
Conversion functions may not have an explicit return type 174
Conversion functions must be member of a class 173
Ctor initializer: "(" expected 175
Data type undefined 167
Declaration expected 166
Declaration: can not use a "( )" initializer here 175
Declarations are not allowed here 165
Default Argument not at end 171
Default parameter has illegal type 170
Definition has illegal Type 166
Destructor declaration: Empty parameterlist "()" expected 175
Destructor name does not match class name 175
Destructor: class name expected after "~" 174
Division by zero 178
Double use of parameter register 171
Each "switch" statement may have only one "default" label 180

Empty source! 174
Error in condition 164
Every reference needs an initializer 182
Expression does not match function return type 180
Expression does not match type of "switch" expression 180
Expressions left and right of ":" need common type 178
Filename expected 163
Friend declaration: "xxx" unknown 166
Function "xxx" has internal linkage and is undefined 181
Function already used 182
Function call: can not use register A4 in small data model 170
Function call: Expression left of () is not a function 170
Function declaration: Illegal storage class 171
Function has illegal return type 166
Function needs return value 179
Function parameters differ from first declaration 165
Function redefined 171
Function templates: Argument "name" never used 184
Functions can not be defined local to functions 165
Header file has wrong version number - please recompile it! 185
Header file is invalid! 185
Identifier "name" redefined 166
Identifier "xxx" not defined 169
Identifier expected 162, 167
Illegal argument 171
Illegal jump into try block 185
Illegal parameter for "operator = ()" 178
Illegal pointer-to-member expression 172
Illegal referenced type 166
Illegal type conversion 170
Illegal use of "friend" specifier 168
Illegal use of Bitfields 168
Illegal use of function specifier 171
Illegal use of storage class specifier 167
Implicit conversion "void*" - "TYPE*": allowed in C Mode only 181
Incomplete type-specification 165
Inconsistent exception specification 184
Inconsistent linkage 171
Inconsistent use of register parameters 181
Initialized object is not an aggregate 169
Initializer for Member "name" expected 175
Initializer has illegal Type 169
Integral or enumerated expression expected 180
Internal error in Code generation 185
Jump past initialization of object "name" 179
Label "name" redefined 179
Label "name" undefined 179
Label expected 179
Library vector Offset ("= short constant") expected 181
Line number expected 164
Linkage specification may only occur in file scope 172
Local function definitions can only be "extern" 172

Macro "name" redefined....162
Macro expansion: "(" expected....162
Macro expansion: Illegal use of "#" or "##"....163
Macro name expected....163
Member "name" not initialized....175
Member function call: class expression has illegal type....172
Member function re-declared....166
Member of Class "xxx" expected....169
Member or baseclass initialized twice....175
Memory overflow!....162
Multiple use of same base class....168
Name of baseclass or nonstatic member expected....175
Name of pointer/int variable expected....181
No "#pragma header" found....185
No access to member "name1" of class "name2"....167
No actual "this" pointer....166
No assignment to vectors, functions, and void expressions....178
No default constructor for class "name"....175
No matching "#if"....164
No matching constructor....175
No such member function in class "name"....184
Old-style parameter declaration expected....180
Operator "delete" must return "void"....176
Operator "delete": 1st parameter must have type "void *"....176
Operator "delete": 2nd parameter must have type "size_t"....176
Operator "delete": too many parameters....176
Operator "new" must return a pointer....176
Operator "new": first operand must be of type "size_t"....183
Operator "new": type expected....176
Operator "sizeof": Type has undefined size....174
Operator or type description expected....173
Operator overload: at least one parameter must be "class"....173
Operator overload: Illegal number of formal parameters....173
Operator requires integral operands....178
Operators "++" and "--" need a l-value....170
Operators may not have default arguments or ellipsis....173
Operators must not be static members....174
Overflow at nesting of Includes (max. depth is 25)....163
Parameter has illegal Type....170
Please switch to C-Mode for old style declarations....166
Pointer expression expected....177
Pointer to member: Class expression expected....172
Pointers/references to references are illegal....165
Preprocessor command expected....162
Pseudo-filename expected....164
Recursive definition of macro....163
Redefinition of default argument....171
Redefinition of default argument for parameter "xxx"....171
Reference conversion: L-value expected....177
Referenced type has unknown size....178
Register objects must be local to a function....168
Return type differs form first declaration....167

Second parameter must be "int" 173
Sorry, but ANSI C does not know member functions 181
Sorry, can't convert virtual base class to derived class 181
String constant exceeds line 162
Struct/class "xxx" undefined 166
Subscribt: int expression in []'s expected 170
Subscript: operand before []'s must be vector or pointer 170
Syntax error 172
Syntax error in #pragma 181
Syntax error in base-list 168
Syntax error in declaration 168
Syntax error in enumeration 167
Syntax error in expression or declarator 168
Syntax error in Parameter list 170
Syntax error in type description 166
Syntax error inside class template "xxx" 183
Syntax error inside inlined member function "xxx" 182
Syntax error inside template function "name" 184
Template argument expected 183
Template argument list differs from previous declaration 183
Template argument list: " " expected 183
Template argument list: "," expected 183
Template argument list: "<" expected 183
Template function redefined 184
Template: class or function declaration expected 184
This operator must be a member function 174
This overloaded function has different access modes 166
This template has already been defined 183
This type does not represent an object 176
Too many errors 164
Too many struct/class members 169
Too many Vector elements given 169
Tried to change functions access 167
Type argument expected 183
Type conflict 172
Type conversion by ambiguous user conversion functions 173
Unary operator "op" used on illegal argument 174
Unary operator used on illegal type 170
Unexpected end of source 182
Unions may not be derived 167
Unions may not have members with Constructors or Destructors 165
Unknown function "xxx" 170
Unknown linkage 172
User Error: 164
Vector element type has undefined size 176
Vector has element type of unspecified size 165
Vectors of references, functions or void are illegal 165
Virtual functions can not be static 177
Virtual functions differ in return type 176-177
Virtual functions must be members of a class 177
Void functions may not return data 180
Workspace overflow 186

Compileroption 474
Configuration Icon 114
Const 172, 178, 305, 512
Container-Typen 491
Containerklassen 623
Continue 179, 240
Copper 755, 758
Copyrights und Warenzeichen V-2
Cos 564
Cosh 564
CPPOPT 81
Ctime 586
ctype.h 262, 557

D

Das Vortext-Symbol 691
Datei 114
Datei speichern 193

Dateioperationen
- Laden 92
- Speichern 91

Datentypen 209, 210, 226, 265
- char 211
- int 210
- long 211
- long int 211
- long long int 211
- short int 211
- signed 211
- signed char 212
- unisgned char 212
- unisgned short int 211
- unsigned int 210
- unsigned long int 211
- unsigned long long int 211

Datestamp 45, 582
Debugger 77
Def-Datei laden 123
Default-Argumente 257
Default-Parameter 253
Default: 180
Define 466
Definitionen 251
Deklaration 316
Deklarationen 251, 292
Dekrement 215, 454
DELETE 87, 177, 334, 337, 443, 447
Der Export von Hilfstexten 693

Der HotHelp-Textmarker 720
Destruktoren 354, 393
Die Bedienung von HotHelp 688
Die Hilfsfunktion der HotHelpTools 704
Die integrierte Suchfunktion 695
Die Kapitelübersicht 693
Die Quelldateien 720
Die Schlüssel 722
Die Texte 720
Die Titelzeile des HotHelp-Fensters 689
Die Übersetzung 723
Difftime 585
Direktive 738
Display Beep 115
Div 530
Dividieren 218
Div_t 530
Do 240
DosBase 738
Double 224, 492
Doubletostr 563
Drag&Drop 662
Drucke Zeilennummern 117

E

EASSERT 560
EasyHotHelp 727
EBNF-Darstellung 118
Editor 85

Editor-Fehlermeldungen
(erwartet 198
) erwartet 198
Alles ist rückgängig gemacht worden 190
Alles ist wiederhergestellt worden 190
Altes Pictogramm ersetzen 192
Auflösungen nur im Bereich von 0 bis 2 195
Automode-Bereich nur von 0 bis 2 196
Cursor steht auf keinem Strukturelement 190
Datei existiert bereits 193
endif ohne if 196
Falsches Argument 199
Fehler beim Lesen 194
Funktion würde Einfaltung zerstören 189
Geänderten Text verwerfen 191
IDENTIFIER erwartet 198
Markierung gesetzt 187
Kann Datei nicht öffnen 194
Kann Info-Datei nicht speichern 194
Kann nicht speichern 194
Kein Befehl 198
Kein Block definiert 188

Kein Pictogramm vorhanden 187
Kein Speicher mehr 191, 193
Konnte Backup nicht erzeugen 188
Konnte Datei nicht speichern 194
Konnte Formular nicht öffnen 195
LISToder IDENTIFIER erwartet 199
LVAL erwartet 199
Markierung nicht gesetzt 188
Menü ist zu groß für den Screen 191
NUMBER erwartet 198
Präprocessor-Befehl zu komplex 196
Präprozessor Befeh erwartet 197
Semikolon erwartet 198
STRING erwartet 198
Strukturen stimmen nicht überein 190
Suchmuster nicht gefunden 189
Tabulatorweite nur zwischen 1 bis 20 195
Text ist geschützt 187
TITLE erwartet 197
Wrap-Position darf nicht kleiner als 10 sein 195
Zeichenkete zu lang 189
Zeiten kleiner 30 Sek. machen keinen Sinn! 196
Zu viele Argumente 199
Zu viele Menüeinträge 197
Zu viele Menütitel 197

Editorkommandos 126
Again 137
AutoMode 150
BackSpace 127
BigStepDown 128
BigStepUp 128
BlockHigher 136
BlockLower 136
BlockSL 136
BlockSR 136
Bot 130
Copy 135
Cut 134
Delete 127
Display 149
Do 149
EditForm 148
EndTextRead 155
EnterCmd 148
ExecCli 156
ExecRexx 151
ExecRexxString 151
Exit 146
Find 137
FindForm 147
FindRepForm 147

Fold 139
Font 150
FreeText 131
FreeWin 145
FullSizeWin 146
GetBlockInfo 153
GetLineString 151
GetScreenInfo 154
GetTextInfo 153
GetTextList 152
GetTextPos 152
GetWindowList 152
GetWord 151
GlobalForm 147
GoBookMark 141
GoDispPos 130
GoMatchStruct 138
GoParseError 138
GoPredFold 140
GoPredKey 140
GoStringPos 130
GoSuccFold 140
GoSuccKey 140
GotoForm 147
Iconify 156
IndentReturn 128
InfoForm 148
InitTextRead 154
InsertChars 156
InsertFile 133
Jump 138
KillLine 127
Mark 134
MarkVert 134
ModifyOldCmd 149
MoveDown 129
MoveLeft 129
MoveRight 129
MoveUp 128
New 132
NewText 131
NewWin 145
Open 133
OpenClip 135
OpenDefs 150
PageForm 147
Paste 135
PasteVert 135
PlayRecScript 144
PredText 131
PredWin 145
Print 132

Quit 146
RecPlay 144
RecStart 144
RecStop 144
ReDo 155
RemFold 139
RemTab 138
RepeatCmd 148
Replace 137
ReplaceAll 137
Return 128
Save 133
SaveClip 135
SaveRecScript 144
ScreenDown 129
ScreenUp 129
SelTextN 131
SetBookMark 141
SetFoldsOff 139
SetFoldsOn 139
SetMode 156
SetOverOff 142
SetOverOn 142
SetTab 142
SetWrap 142
SetWrapOff 143
SetWrapOn 142
ShowFold 139
SplitHorWin 146
SplitVerWin 146
SuccText 131
SuccWin 145
TextRead 155
ToggleKey 140
Top 130
UnDo 155
UndoLine 127
UpdateText 133
UpdateTexts 134
WrapPara 143
WrapText 143

Edward 85
Edward.DEF 4
EdwardCommand 119
EFREEMEM 560
Eigenschaften von HotHelp 681
Einfügen 93
Einstellungen 112
Einteilung in Einträge 721
Ellipse 173, 258, 430, 434, 515, 572, 591
Ellipsen-Parameter 434

Else 165, 179
ENDCOP 758
Ende 111
Enforcer 657
ENOMEM 560
ENONUM 560
Entferne Falte 101
Enum 326
EOF 537
EOF 540
ERANGE 529, 559
Errno 559
Ersetzen 98
Erstelle Backup 115
Erzeuge Falte 101
EUSRBRK 559
Exception Handling 509
Exceptions 28
Exit 475, 532, 534, 555
Exp 564
Expo10 567
Exponent 225
Extern 171

F

Fabs 565
Fallunterscheidungen 230
Falten 100
Falten überspringen 117
Farben 113
Fclose 538
Fehler 77
Fehlererkennung 596
Fehlermeldungen 160

Fenster
- Erzeugen 107
- Mit Ansichten arbeiten 109
- Schließen 108
- Wählen 107

Feof 549
Ferror 549
Fflush 551
FFP-Single 3
Fgetc 540
Fgetpos 555
Fgets 541
FILE 536
FILENAME_MAX 539
Fließkommaoperationen 226
Fließkommatypen 224

Float 224-225
Floattostr 563
Floor 565
Fmod 566
Fontgröße 113
Fontname 113
Fopen 538
FOPEN_MAX 539
For 179, 240
Form 591
Form Feed 117
Fpos_t 554
Fprintf 545
Fputc 541
Fputs 542
Fpwr10 567
Fread 553
Free 532
Freopen 539
Frexp 566
Friend 168
Fscanf 548
Fseek 554
Fsetpos 555
Ftell 554
Füllbytes 747

Funktionen 244, 657, 743
- Deklaration 251
- Ellipse 257
- Gültigkeitsbereiche 249
- main 244
- Member-Funktionen 355
- Rekursive 251
- Rückgabe 247
- Überladen 253
- Variable Parameterliste 257
- Vektoren als Parameter 277

Funktions-Templates 498
Funktionsparameter 267
Fuß 116
Fwrite 553

G

Gadgets
- BOOPSI 630
- GadTools 630

GadTools Gadgets 630
Gehe zu Fehler 99
Gehe zur Zeile 99

GetBaseReg 615
Getchar 540
Getenv 535
Gets 541
Getstream 596
Gleitpunktzahl 225
Gmtime 583
Goto 179, 185, 238, 567
Grafiken 723
Großbuchstaben 96
Grundlagen der Projekterstellung 718
Gültigkeitsbereiche 233, 249, 294, 326

H

Hello World 204
Hexadezimal 213
Hilfe über Hilfe 696
HotHelp 680
HotHelpComp - der HotHelp-Übersetzer 728
HotHelpMarker 718
HotHelpPref - HotHelps Voreinstellungen 706
HotHelpPro - Der Projektmanager 714
HotHelpTools 704
HotKey 111, 123, 688
HUGE_VAL 529
Hunks 68
Hypertext 680

I

Iconify 111
IDCMP-Flags 627
IEEE-Single 3
If 165, 179
Implementation 382
Include 470, 754
Information Hiding 358
Initialisieren 343

Initialisierung 336
    von Klassen 393

Initialisierungen 229
InitModules 65, 615
Inkrement 215, 454
Inline 171, 310, 358, 495, 671
Inline-Funktionen 309
Int 210
Intelligente Zeiger 453
IntMult 32 218
Inttostr 563
IOFBF 550

IOLBF 550
IONBF 551
Isalnum 557
Istream 437

J

Jederzeit hilfsbereit 680
Jmp_buf 570
JMP_BUF_SIZE 570

K

Kapitel, Vorgänger und Nachfolger 725
Kernigham & Richie 258
KeyDef 121
Keys 121
Kickstart 658
Klammern Einzug 104

Klasse
- ArgsMCursorC 648
- BufferC 647
- CurrentDirC 646
- DirLockC 646
- EListC 623
- Exception 623
- FileHandleC 647
- FileLockC 646
- GadgetC 630
- GadgetListC 630
- GagdteEventC 630
- GCycleC 630
- gen_array 623
- gen_arraylist 623
- GeometryC 633
- GeometryGroupC 633
- GModelC 630
- GroupLayoutC 632
- GTIDCMPortC 630
- GTWindowC 630
- HandlerPortC 626
- IDCMPortC 629
- InputHandlerC 647
- ios 596
- LayouC 632
- LayouterWindowC 644
- LBBevelboxC 643
- LBBevellineC 644
- LibraryBaseErrC 621
- ListC 623
- MessageHandlerC 626
- OutputHandlerC 647

PortC 625
ResourceX 623
RootGeometryC 639
SignalsC 625
StandardWindowC 619, 645
String 606
TextAttrC 645
WindowC 627
WindowCloseHandlerC 629
WindowEventHandlerC 628
WindowSnapshotC 628

Klassen 353
Klassen-Templates 491
Klassenbibliothek 3
Kleinbuchstaben 96
Kommandozeilen-Interpreter 112
Kommandozeilenargumente 285
Kommandozeilenversion 474
Kommentar 734
Kommentare 54, 204
Konfigurationsdatei 4

Konstante 212
Suffix
l 212
LL 213
U 212
ULL 213

Konstanten 751
Konstruktoren 354, 393-394
Konvertierungsfunktionen 423
Kopf 116
Kopieren 93

L

L-Wert 215-216, 223
Label 738
Labelfeld 734
Labs 529
Laden 117
Laufzeitfehler 517
Ldexp 566
Ldiv_t 530
Lesezeichen 102, 682
Lexikon 680
Linker 59, 79, 158
Linker Rand 117
Linker-Bibliotheken 63
Literale 212, 224
Lizenzbedingungen V-3

LN_SUCC 738
Localtime 584
Log 512, 564
Log10 564
Logfile 60, 63
Long 211
Long double 224
Long int 211
Long long int 211
Longjmp 517, 567

M

MACRO 742
Main 205, 244
Make 13, 43
Makros 105
Malloc 337, 532
Mantisse 225
Markieren 93
Markieren der Texte 720
math.h 263

MaxonSDB
- Menü Einstellungen
  - Einstellung laden 670
  - Einstellung sichern 670
  - Einstellung sichern als 670
- Menü Projekt
  - Debugger beenden 669
  - Öffnen 669
  - Über 669
  - Wieder Starten 669
- Menü Schritt
  - Neue Adresse zeigen 669
  - Programm abbrechen 670
- Programmstart 659
- Vorbereitung 659

MCPPC 69
Mehrfach-Anweisungen 54
Member 420
Member-Funktionen 355, 377, 495
Memchr 582
Memcmp 582
Memcpy 159, 581
Memmove 159, 581
Memset 582
Menu 119
MenuGroup 119
MenuItem 120
Mktime 584
MMU 735

Mnemonics 734
ModeQualList 121
Modf 566
Modul 657
Möglichkeiten, HotHelp aufzurufen 697
Multiplizieren 218

N

NARG 739
Neuer Font 113
New 176, 334, 337, 443-444
Nil: 535
No errors 185
NULL 533, 537, 575
Nummernbereich 117

O

Objekte 353
Offsetof 575
Oktal 213
Onlineprojekte 681, 699, 715
Onlineprojekte und andere Editoren 700
Opcodefeld 734
Operandenfeld 734

Operator
- ! 597
- & 182
- && 597
- || 597
- ## 470
- * 453
- + ... 461
- delete 447
- Funktionsaufruf 450

Operatoren 214, 437
- ++ 454
- += 157
- -- 454
- = ... 441
- ?: 222
- Komma-Operator 224
- Linksshift 219
- Logische
  - && 221
  - || 221
- Multiplikative
  - % 217
  - * 217
  - / 217

Postfix
++ 215
-- 215
Shift
<< 218
>> 219
unäre 229
Unäre
! 216
+ 216
++ 215
- 216
-- 215
sizeof 216
~ 216
Zuweisungsoperatoren 223
|= 157

ORG 750
Ostream 437, 591
Overload 255

P

Parameter 245, 657
Parameterliste 246
Parameterlisten 259
PARAMETER_BASE 562
Perror 555
Pfad 115
Pointer 265
Pow 565

Preprozessor 73, 85, 157, 205, 301, 465, 480
Arbeitsweise 482

Preprozessoranweisung 470
Printf 542
Private 167
Programmausführung 474
Projekt & Ende 110
Projekt- und ARexx-Querverweise 725
Projekttest 716

Projektverwaltung 45
Abhängigkeiten 48
Änderungen per Hand 52
Compileranweisungen 50
Dateien hinzufügen 48
Kommentare 54
Makros 55
Projektpflege 49
Regeln 56

Übersetzen ... 48
Übersetzung erzwingen ... 50
wechseln ... 50
Zeilenverbindung ... 54

Protected ... 374
Prototyp ... 252
Prozessor-Exception ... 225
Ptrdiff_t ... 575
Public ... 167
Putchar ... 542
Puts ... 542
Pwr10 ... 566

Q

Qsort ... 491, 535
Qualifizierungen ... 292
Querverweise ... 690, 722
QuotedString ... 119

R

Raise ... 571
Rand ... 531
Random ... 531
RAND_MAX ... 531
Realloc ... 533
Rechter Rand ... 104, 117
Redo ... 124
Referenzen ... 265, 288
Register ... 168, 171, 182, 301
Registervariablen ... 301
Rekursiv ... 251
Remove ... 552
Rename ... 552
Reservierte Symbole ... 739
Return ... 160, 180
Rewind ... 554
Rollsymbole ... 690
RSRESET ... 754
RSSET ... 754
Rückgabewert ... 205

S

Scanf ... 546
Schleifen ... 235
Schlüssel, Projekte und Muster ... 691
Schlüsselgruppen ... 725

Schlüsselwort ... 207, 492, 519, 743, 749
class ... 492

template 492

Schriftarten 723
Scope2 ..... 33, 239, 326, 360
Scoperegeln 373
Scrollbereich 115
SECTION-Befehl 749
Seitenformat 116
Semaphors 71
Setbuf 551
Setjmp 517, 567
SetMode 121
Setvbuf 550
Set_new_handler 532
Set_terminate 606
Set_unexpected 605
Shellstart 660
Shift 219
Short int 211
Short-Cuts 112
Signed 211
Signed char 212
SIGTERM 571
Sig_atomic_t 572
SIG_DFL 571
SIG_ERR 571
SIG_IGN 571
Sin 564
Sinh 564
Sizeof 174, 216, 575
Size_t 183, 532-533, 537, 575, 602
Speicherklassen 292
Speichern 117
Sprachstandard 157
Sprintf 545
Sprünge 238
Sqrt 564
Srand 531
Sscanf 549
Stackpointer 739

Standardfunktionen 261
- abs 264
- ceil 263
- cos 263
- exit 263
- exp 263
- fabs 263
- floor 263
- Isalpha 262
- islower 262
- isupper 262

labs 264
log 263
printf 262
qsort 314
rand 264
Random 264
sin 263
sqrt 263
tan 263
tolower 263
toupper 263

Standardprojekte 714
Start der HotHelpTools 705
Startup-Code 63, 65
Static 171, 181, 296, 310, 496, 521
Statische Daten 292
Statische Member 364
Status-Anzeige 86
Stderr 537
Stdin 537
Stdio.h 262
stdlib.h 263
Stdout 537
Std__err 536
Std__in 536
Std__out 536
Steuerzeichen 213
Strcat 576
Strchr 578
Strcmp 577
Strcpy 286, 576
Strcspn 579
STREAM_MAXSTRING 541
Strerror 579
Strftime 585
Stricmp 577
string.h 262
Strings 278
Stringvergleiche 281
Strlen 285, 579
Strlwr 578
Strncat 577
Strncmp 577
Strncpy 576
Stroustrup 514
Strpbrk 579
Ströme 597
Strrchr 578
Strspn 578
Strstr 579
Strtod 528

Strtok 580
Strtol 527
Strtoul 527
Strtouvl 528
Strtovl 528
Struct 323
Struktur-Check 98

Struktur
List 623
MinList 623

Strukturen 321
Strukturieren 100
Strukturkontrolle 99
Strukturoffsets 754
Strupr 578
Subklasse 367
Subtraktion 218
Suchen 96-97
Suffix 212
Superklasse 367
Support V-2
Switch 180, 240, 242
Symbole 480
Syntax 312
System 535

T

TAB in Spaces 103
TAB Weite 104
Tan 512, 564

Tastaturbedienung
HotHelp 697

Templateargument 502
Templates 184, 491
Temporäre Objekte 426

Text
Auswahl 89
Drucken 90
Erzeugen 89
Freigeben 90
Löschen 90

Textdraw 115
Throw 28, 158, 185, 519
Time 583
Time_t 582
Titelzeile 86

Tm 583
Tmpfile 553
Tmpnam 552
Tolower 558
Tooltypes 623
Toupper 558
Trigraph 477, 482
Trigraph-Sequenzen 214
Try 28, 158, 519
Try-Blocks 515
Typdefinitionen 304
Typedef 304, 602
Typen 224
Typregeln 371
Typsicheres Linken 297

Typumwandlung 228
casting 228

U

Überladen 253, 429
new 444

Überschreiben 103
Übersetze CRs 104
Übersetzung 471
UCHAR_MAX 562
Uinttostr 563
Undo 124
Undo Konfiguration 114
UndoLine 87
Unexpected 522, 605
Ungetc 540
Unionen 328
Unsigned char 212
Unsigned int 210
Unsigned long int 211
Unsigned long long int 211
Unsigned short int 211
Uvlongtostr 563

V

Variable Parameterlisten 257
Va_arg 573
Va_end 574
Va_list 572
Va_start 573
Vektordeklarationen 275

Vektoren 265, 271-272

dynamisch erzeugt ... 335
Initialisierung ... 286

Vektorkonzept ... 491
Vererbung ... 371
Vererbungslehre ... 366
Vergleiche ... 220

Verknüpfungen
Bitweise ... 221

Verschiedene Ansichten über HotHelp ... 694
Vertauschen ... 98
Vertikal einfügen ... 94
Vertikal markieren ... 94
Verwendung von Lesezeichen ... 696
Verzweigungen ... 230
Vfprintf ... 545
Vielfachverzweigungen ... 242
Virtual ... 171
Virtuelle Member ... 355
Vlabs ... 530
Vlongtostr ... 563
Void ... 160, 166, 181, 205
Void* ... 176
Volatile ... 172, 305, 308, 569

Vordefinierte Symbole
__COMPMODE__ ... 481
__cplusplus ... 481
__DATE__ ... 480
__FILE__ ... 480
__LINE__ ... 480
__MAXON__ ... 481
__STDC__ ... 481
__TIME__ ... 480

Vorzeichen ... 225
Vprintf ... 545
Vsprintf ... 545

W

Warnungen ... 77, 160
Funktion „name“ has no return statement ... 160
Nested comment ... 160
No prototype for function „name“ ... 160
Precompiled header file "name" not found ... 161
Statement has no effect at all ... 160
Suspect "=" in condition ... 161
Temporary object for non-const reference introduced ... 161
This seems to be an old-style parameter list ... 161
Unknown #pragma (maybe misspelled?) ... 161

Unsafe type conversion 161
Variable name never initialized 161
Variavle xxx declared but never used 160
Very long argument for Macro "xxx" - probably missing ")" 162

Wbparse 159
Wchar_t 534, 576
Which_xdigit 558
While 158, 240
Wildcard 97
Word Wrap 103-104
Workbenchargumente 623
Wortsymbol 209

Z

Zahlenformate 740
Zeige Falte 101
Zeige Spaces 103
Zeige TABS 103

Zeiger 265, 272, 420
auf Funktionen 311
Überladene Funktionen 313

Zeigertypen 268
Zeilen bis zum Text 116
Zeilen nach dem Text 116
Zeilen/Seite 116
Zeilen/Text 117
Zeilenverbindung 54
Zeit 115
Zugriffsrechte 358
Zum Anfang 116
Zum Ende 117
Zuweisungs-Operator 441

Maxon C++ 3

Features

Light-Version
- C++-Compiler (AT&T 2.0 Standard)
- ANSI C-Compiler (Aztec- und Lattice-ANSI-kompatibel)
- CLI-Compiler
- HotHelp 3-Reader zum Anzeigen von HotHelp-Projekten

Developer-Version
- C++-Compiler (AT&T 3.0 Standard)
- Templates, Exceptions
- ANSI C-Compiler (wie Light-Version)
- CLI-Compiler
- beinhaltet zusätzlich: MaxonHotHelp 3, MaxonSDB, MaxonEASY-OBJECTS und MaxonASM-CLI

Compiler (C/C++)
- komfortable Projektverwaltung (Make)
- direkter Fehlerstellenansprung
- voll in den Editor integriert
- über ARexx-Schnittstelle steuerbar
- sehr schnelle Turnaroundzeiten
- echte vorkompilierte Include-Dateien (Preincludes)
- Include-Dateien für Kickstart 3.1
- speicherbarer Programmstatus
- Codegenerator für prozessorspezifischen Code 68010/20/30 und 68881/2 FPU

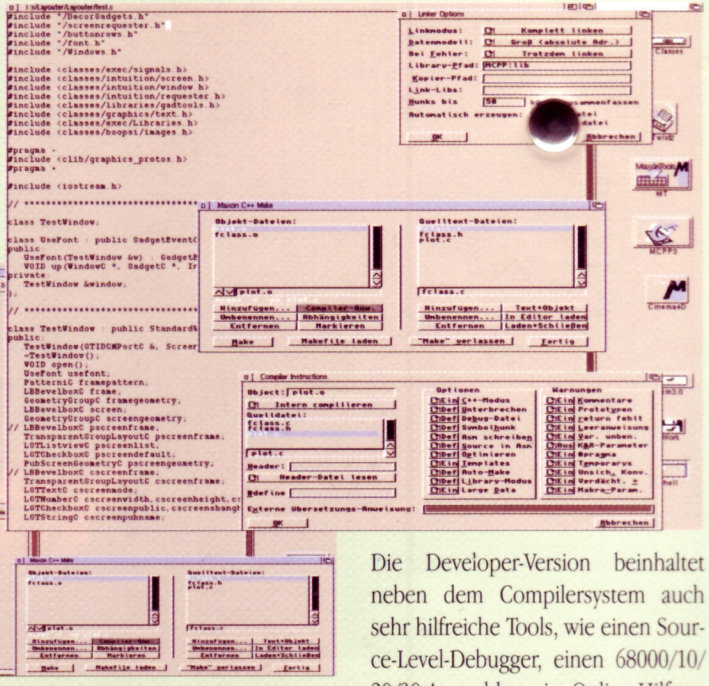
Programmeditor (EDWARD)
- Multi-Window-Technik
- beliebig viele Texte gleichzeitig bearbeitbar (speicherabhängig)
- Tastatur- und Menüfunktionen frei belegbar
- speicherbarer Programmstatus
- ARexx-Schnittstelle

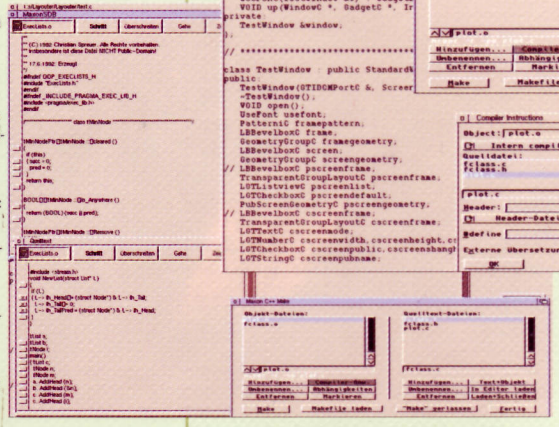
Hilfesystem HotHelp 3
(nur in der Developer-Version)
- Deutschsprachige Beschreibung der Library- und Device-Funktionen zu Amiga-OS 2.1
- jederzeit aktivierbar
- kontextsensitiv
- komfortabler Editor für eigene Projekte
- Online-Projekte (direkte übernahme von Texten und Sourcecode in das Hilfesystem)
- ARexx-Schnittstelle
- Grafikeinbindung direkt im Hilfetext
- direktes Lesen von AmigaGuide™-Projekten
- lizenzfreie Weitergabe des HotHelp 3-Anzeigers

Features der Klassenbibliothek (MaxonEASY-OBJECTS)
(nur in der Developer-Version)
- Ressourcenhandling: Betriebssystemressourcen (Speicher, Signale, Fenster, Bildschirme etc) werden automatisch verwaltet und bei Programmende freigegeben (keine aufwendige Fehlerbehandlung mehr nötig).
- Fehlerbehandlung durch Exceptions
- Handlerkonzept
- Datenstrukturen: Listen, dynamisch große Arrays, Datenpuffer (z.B. für I/O-Operationen)
- Verwaltung folgender Betiebssystemresourcen: Exec: Signale, Nachrichtenports, Speicher, Libraries, Devices, Listen - DOS: Locks, Dateien, Pfadnamen, Kommandozeilenargumente, Notifikation - Utility: Hooks und Taglisten - Intuition: Bildschirme, Fenster, IDCMP-Nachrichten, Systemrequester - Libraries: GadTools, ASL, Icons, Workbench, Tooltypes, Applikationenfenster
- BOOPSI Unterstützung
- automatisch größenveränderbare und font-veränderbare Fenster durch den Layouter
- deutschsprachige Online-Dokumentation

Externer Assembler (MaxonASM-CLI)
(nur in der Developer-Version)
- 68000/10/20/30 Makro-Assembler
- FPU 68881/2 und MMU-Unterstützung
- Multi-Pass-Optimierung erzeugt sehr schnellen und kompakten Programmcode.
- voll in die Projektverwaltung integrierbar

C/C++ Source Level Debugger (MaxonSDB)
(nur in der Developer-Version)
- Multi-Window-Technik
- einfache Bedienung - Drag&Drop
- Beobachten von Variableninhalten
- Haltepunkte
- direkt vom Compiler aus aufrufbar





MaxonC++ ist die erste konsequente Implementierung des C++ 3 Standards auf dem Amiga.

Das MaxonC++-Compilersystem beinhaltet sowohl einen ANSI C-, als auch einen zum AT&T 3.0-Standard kompatiblen C++-Compiler. Der Compiler erzeugt aus den C++-Quelltexten direkt ausführbare Programme ohne Umweg über C, Assembler oder andere Sprachen. Durch die volle Systemintegration werden überzeugend schnelle Turnaroundzeiten erreicht.

Die Developer-Version beinhaltet neben dem Compilersystem auch sehr hilfreiche Tools, wie einen Source-Level-Debugger, einen 68000/10/20/30-Assembler, ein Online-Hilfesystem mit der deutschsprachigen Beschreibung der Systemfunktionen und eine umfangreiche Klassenbibliothek.

Das auch in der Light-Version aufwendige und gut strukturierte deutschsprachige Handbuch ist ein kompletter ANSI-C/C++-Programmierlehrgang und auch für Einsteiger zu empfehlen.

Lauffähig auf allen Amigas mit min. 2 MB Speicher und Amiga-OS 2.0. Bei Nutzung der Klassenbibliothek werden 6 MB Speicher benötigt.

AMIGA®

4 010509 411302

MAXON Computer GmbH • Industriestr. 26
D-65760 Eschborn • Telefon 06196/481811
Irrtümer und Änderungen vorbehalten!

MAXON computer